ISO/IEC JTC 1/SC 22/OWGV N 0125

Editors's draft of PDTR 24772 Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use, 25 March 2008

Date	26 March 2008
Contributed by	John Benito, Editor
Original file name	PDTR24772-308.pdf
Notes	Replaces N0106

ISO/IEC JTC 1/SC 22 N 0000

Date: 2008-03-26

ISO/IEC PDTR 24772

ISO/IEC JTC 1/SC 22/OWG N0125

Secretariat: ANSI

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

Élément introductif — Élément principal — Partie n: Titre de la partie

Document type: International standard
Document subtype: if applicable
Document stage: (40) approval stage
Document language: E

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772, which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Programming Languages.

Introduction

A paragraph.

The **introduction** is an optional preliminary element used, if required, to give specific information or commentary about the technical content of the document, and about the reasons prompting its preparation. It shall not contain requirements.

The introduction shall not be numbered unless there is a need to create numbered subdivisions. In this case, it shall be numbered 0, with subclauses being numbered 0.1, 0.2, etc. Any numbered figure, table, displayed formula or footnote shall be numbered normally beginning with 1.

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

1 Scope

1.1 In Scope

The technical report specifies software vulnerabilities that are applicable in a context where assured behaviour is required for security, safety, mission critical and business critical software, as well as any software written, reviewed, or maintained for any application.

1.2 Not in Scope

This technical report does not address software engineering and management issues such as how to design and implement programs, using configuration management, managerial processes etc.

The specification of an application is *not* within the scope.

1.3 Approach

The impact of the guidelines in this technical report are likely to be highly leveraged in that they are likely to affect many times more people than the number that worked on them. This leverage means that these guidelines have the potential to make large savings, for a small cost, or to generate large unnecessary costs, for little benefit. For these reasons this technical report has taken a cautious approach to creating guideline recommendations. New guideline recommendations can be added over time, as practical experience and experimental evidence is accumulated.

A guideline may generate unnecessary costs include:

1) Little hard information is available on which guideline recommendations might be cost effective
2) It is likely to be difficult to withdraw a guideline recommendation once it has been published
3) Premature creation of a guideline recommendation can result in:
    i. Unnecessary enforcement cost (i.e., if a given recommendation is later found to be not worthwhile).
    ii. Potentially unnecessary program development costs through having to specify and use alternative constructs during software development.
    iii. A reduction in developer confidence of the worth of these guidelines.

1.4 Intended Audience

The intended audience for this document is those who are concerned with assuring the software of their system, that is, those who are developing, qualifying, or maintaining a software system and need to avoid vulnerabilities that could cause the software to execute in a manner other than intended.

As described in the following paragraphs, developers of applications that have clear safety, security or mission criticality are usually aware of the risks associated with their code and can be expected to use this document to ensure that *all* relevant aspects of their development language have been controlled.

That should not be taken to mean that other developers can ignore this document. A flaw in an application that of itself has no direct criticality may provide the route by which an attacker gains control of a system or may otherwise disrupt co-located applications that are safety, security or mission critical.

It would be hoped that such developers would use this document to ensure that common vulnerabilities are removed from all applications.

1.4.1 Safety-Critical Applications

Users who may benefit from this document include those developing, qualifying, or maintaining a system where it is critical to prevent behaviour which might lead to:

- loss of human life or human injury
- damage to the environment

and where it is justified to spend additional resources to maintain this property.

1.4.2 Security-Critical Applications

Users who may benefit from this document include those developing, qualifying, or maintaining a system where it is critical to exhibit security properties of:

- Confidentiality
- Integrity, and
- Availability

and where it is justified to spend additional money to maintain those properties.

1.4.3 Mission-Critical Applications

Users who may benefit from this document include those developing, qualifying, or maintaining a system where it is critical to prevent behaviour which might lead to:

- loss of or damage to property, or
- loss or damage economically

1.4.4 Modeling and Simulation Applications

Programmers who may benefit from this document include those who are primarily experts in areas other than programming but need to use computation as part of their work. Such include scientists, engineers, economists, and statisticians. They require high confidence in the applications they write and use due to the increasing complexity of the calculations made (and the consequent use of teams of programmers each contributing expertise in a portion of the calculation), due to the costs of invalid results, or due to the expense of individual calculations implied by a very large number of processors used and/or very long execution times needed to complete the calculations. These circumstances give a consequent need for high reliability and motivate the need felt by these programmers for the guidance offered in this document.

1.5 How to Use This Document

1.5.1 Writing Profiles

[*Note*: Advice for writing profiles was discussed in London 2006, no words]

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Language Vulnerability

A *property* (of a programming language) that can contribute to, or that is strongly correlated with, application vulnerabilities in programs written in that language.

Note: The term "property" can mean the presence or the absence of a specific feature, used singly or in combination. As an example of the absence of a feature, encapsulation (control of where names may be referenced from) is generally considered beneficial since it narrows the interface between modules and can help prevent data corruption. The absence of encapsulation from a programming language can thus be regarded as a vulnerability. Note that a property together with its complement may both be considered language vulnerabilities. For example, automatic storage reclamation (garbage collection) is a vulnerability since it can interfere with time predictability and result in a safety hazard. On the other hand, the absence of automatic storage reclamation is also a vulnerability since programmers can mistakenly free storage prematurely, resulting in dangling references.

3.2 Application Vulnerability

A security vulnerability or safety hazard, or defect.

3.3 Security Vulnerability

A weakness in an information system, system security procedures, internal controls, or implementation that could be exploited or triggered by a threat.

3.4 Safety Hazard

Should definition come from, IEEE 1012-2004 IEEE Standard for Software Verification and Validation, 3.1.11, IEEE Std 1228-1994 IEEE Standard for Software Safety Plans, 3.1.5, IEEE Std 1228-1994 IEEE Standard for Software Safety Plans, 3.1.8 *or* IEC 61508-4 and ISO/IEC Guide 51*?*

3.5 Safety-critical software

Software for applications where failure can cause very serious consequences such as human injury or death.

3.6 Software quality

The degree to which software implements the needs described by its specification.

3.7 Predictable Execution

The property of the program such that all possible executions have results which can be predicted from the relevant programming language definition and any relevant language-defined implementation characteristics and knowledge of the universe of execution.

Note: In some environments, this would raise issues regarding numerical stability, exceptional processing, and concurrent execution.

Note: Predictable execution is an ideal which must be approached keeping in mind the limits of human capability, knowledge, availability of tools etc. Neither this nor any standard ensures predictable execution. Rather this standard provides advice on improving predictability. The purpose of this document is to assist a reasonably competent programmer approach the ideal of predictable execution.

4 Symbols (and abbreviated terms)

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 build vulnerabilities into software by failing to understand the expected behavior (the software requirements), or by failing to correctly translate the expected behavior into the actual behavior 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, nor 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 combines in sequence language features (functional pieces) expressed in the programming language so the cumulative effect is a written expression of the software's behavior.

A program's expected behavior 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 make the resulting 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 hierarchal 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 is often estimated by the number of lines of code (automatically counted without regard to language statements).

5.1 Issues arising from lack of knowledge

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 than 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[1]. 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 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 behavior which is not specified completely. In any circumstance, of course, a particular compiler will produce a program with some specific behavior (or fail to compile the program at all). Where a programming language is insufficiently well defined, different compilers may differ in the behavior of the resulting software. 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 behavior. 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 behavior not specified by the standard has been produced. Programs relying upon behavior 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 behavior), 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.

5.1.2 Issues arising from implementation defined behaviour

In some situations, a programming language standard may specifically allow compilers to give a range of behavior 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 completely the size of storage reserved for language-defined entities. The degree to which a diligent programmer may inquire of the storage size reserved for entities varies among languages.

The authors of language standards are encouraged to provide lists of all allowed variation of behavior (as many already do). Such a summary will benefit applications programmers, those who define applications coding standards, and those who make code-checking tools.

[1] This feature has been deprecated in the 1999 version of the ISO C Standard.

5.1.3 Issues arising from undefined behaviour

In some situations, a programming language standard may specify that program behavior 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.

Generally, the amount of resources available to a program (memory, file storage, processor speed) is not specified by a language standard. The form of file names acceptable to the operating system is not specified (other than being expressed as characters). The means of preparing source code for execution may be unspecified by a language standard.

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 behavior 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 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 which 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 Portability

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 behavior is indicated by a factor that is not a part of the source code. That is, the program's behavior 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 behavior), 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 behavior. 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 behavior specification beyond the source code. Again, these issues are software engineering issues and are not further discussed in this document.

6. Programming Language Vulnerabilities

The standard for each programming language provides definitions for that language's constructs. This Technical Report will in general use the terminology that is most natural to the description for each individual vulnerability, relying upon the individual standards for terminology details. In general, the reader should be aware that "method", "function", and "procedure" can denote similar constructs in different languages; as can "pointer" and "reference". Situations described as "undefined behavior" in some languages are known as "unbounded behavior" in others.

6.1 FLC Numeric Conversion Errors

6.1.0 Status and history

PENDING
2008-01-04, Edited by Robert C. Seacord
2007-12-21, Merged XYE and XYF
REVISE: Robert Seacord
2007-10-01, OWGV Meeting #6
2007-08-05, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.1.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types [7]:

```
aVar := anExpression
value1 + value2
foo(arg1, arg2, arg3, … , argN)
```

Type conversion seeks to follow these exact match rules while allowing programmers some flexibility in using values such as: structurally-equivalent types in a name-equivalent language, types whose value ranges may be distinct but intersect (for example, subranges), and distinct types with sensible/meaningful corresponding values (for example, integers and floats). Explicit conversions are called *type casts*. An implicit type conversion between compatible but not necessarily equivalent types is called *type coercion*.

Numeric conversions can lead to a loss of data, if the target representation is not capable of representing the original value. For example, converting from an integer type to a smaller integer type can result in truncation if the original value cannot be represented in the smaller size and converting a floating point to an integer can result in a loss of precision or an out-of-range value.

6.1.2 Cross reference

CWE:
192. Integer Coercion Error
CERT C: INT02-A, INT08-A, INT31-C
CERT C++: INT02-A, INT31-C
MISRA C 2004, Rule 12.9

6.1.3 Categorization

Group: Arithmetic

6.1.4 Mechanism of failure

Numeric conversion errors can lead to a number of safety and security issues. Typically, conversion errors in data integrity issues, but may also result in safety and security vulnerabilities.

Numeric values within a typical operational range can be safely converted between data types. Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. These can result in safety issues, for example, the failure of the Ariane 5 launcher which occurred due to an improperly handled conversion error resulting in the processor being shutdown [3].

Conversion errors can also result in security issues. An attacker may input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value may then be used as an array index, a loop iterator, a length, a size, state data, or in some other security critical manner. For example, a truncated integer value may be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow [6].

Numeric type conversion errors often lead to undefined states of execution resulting in infinite loops or crashes. In some cases, integer type conversion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code. Integer type conversion errors result in an incorrect value being stored for the variable in question.

6.1.5 Applicable language characteristics

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

- Languages that perform implicit type conversion (coercion).
- Languages that are weakly typed. Strongly typed languages do a strict enforcement of type rules because all types are known at compile time.
- Languages that support logical, arithmetic, or circular shifts on integer values. Some languages do not support one or more of the shift types.
- Languages that do not generate exceptions on problematic conversions.

6.1.6 Avoiding the vulnerability or mitigating its effects

To protect against corruption of memory, integer values used in any of the following ways must be correct:

Integer values that originate from untrusted sources must be guaranteed correct if they are used in any of the following ways [1]:

- as an array index
- in any pointer arithmetic
- as a length or size of an object
- as the bound of an array (for example, a loop counter)
- in security or safety critical code
- as a argument to a memory allocation function

For dependable systems, all value faults must be avoided.
Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. However, it is difficult to guarantee that multiple input variables cannot be manipulated to cause an error to occur in some operation somewhere in a program [6].

- An alternative or ancillary approach is to protect each operation. However, because of the large number of integer operations that are susceptible to these problems and the number of checks required to prevent or detect exceptional conditions, this approach can be prohibitively labor intensive and expensive to implement.

- A language which generates exceptions on erroneous data conversions might be chosen. Design objects and program flow such that multiple or complex casts are unnecessary. Ensure that any data type casting that you must used is entirely understood to reduce the plausibility of error in use.

Verifiably in range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. Faced with a numeric conversion error, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce a numeric value that is within the range of representable values on that system. The latter semantics may be preferable in some situations in that it allows the computation to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what numeric result to return to the user.

A recent innovation from ISO/IEC TR 24731-1 [8] is the definition of the **`rsize_t`** type for the C programming language. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like **`size_t`**. Also, some implementations do not support objects as large as the maximum value that can be represented by type **`size_t`**.

For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that **`RSIZE_MAX`** be defined as the smaller of the size of the largest object supported or **`(SIZE_MAX >> 1)`**, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define **`RSIZE_MAX`** as **`SIZE_MAX`**, which means that there is no object size that is considered a runtime-constraint violation.

6.1.7 Implications for standardization

6.1.8 Bibliography

[1] CERT. *CERT C Secure Coding Standard*. https://www.securecoding.cert.org/confluence/x/HQE (2007).
[2] CERT. *CERT C++ Secure Coding Standard*. https://www.securecoding.cert.org/confluence/x/fQI (2007).
[3] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.
[4] Hatton 2003
[5] MISRA Limited. "MISRA C: 2004 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, October 2004 (ISBN 095241564X).
[6] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See http://www.cert.org/books/secure-coding for news and errata.
[7] John David N. Dionisio. Type Checking. http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf
[8] ISO/IEC TR 24731-1. *Extensions to the C Library, — Part I: Bounds-checking interfaces*. Geneva, Switzerland: International Organization for Standardization, April 2006.

6.2 OTR Subprogram Signature Mismatch

[For the convenience of reviewers, I have paraphrased the relevant rules from JSF C++:
[108: Functions with variable numbers of arguments are forbidden.
[110: Avoid functions with more than 7 arguments.]

[For the convenience of reviewers, I have paraphrased the relevant rules from MISRA 2004 below:
[8.1: Provide prototype declarations for functions that are visible at both the function definition and the call of the function.
[8.2: Declare and/or define the type of any function.
[8.3: The type of each parameters and the return type must be identical in the function declaration and definition.
[16.1: Do not define functions that take a variable number of arguments.
[16.3: Give names to all of the parameters in a function prototype declaration.
[16.4: Use the same parameter names in the declaration and definition of a function.
[16.5: Functions that don't take any parameters must be declared with a parameter type of void. Same for the return type.
[16.6: Always pass the same number of arguments to a function as appear in the prototype declaration and the definition.]

6.2.0 Status and history

2007-12-21, Jim Moore: Drafted as a merger of XYG and XZM.

6.2.1 Description of application vulnerability

If a subprogram is called with a different number of parameters than it expects, or with parameters of different types than it expects, then the results will be incorrect. Depending on the language, the operating environment, and the implementation, the error might be as benign as a diagnostic message or as extreme as a program continuing to execute with a corrupted stack. The possibility of a corrupted stack provides opportunities for penetration.

6.2.2 Cross reference

CWE:
230. Missing Value Error
231. Extra Value Error
234. Missing Parameter Error
MISRA 2004: 8.1, 8.2, 8.3, 16.1, 16.3, 16.4, 16.5, 16.6
JSF C++: 108, 110[?]

6.2.3 Categorization

See clause 5.?. *<Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>*

6.2.4 Mechanism of failure

When a subprogram is called, the actual arguments of the call are pushed on to the execution stack. When the subprogram terminates, the formal parameters are popped off the stack. If the number and type of the actual arguments does not match the number and type of the formal parameters, then the push and the pop will not be commensurable and the stack will be corrupted. Stack corruption can lead to unpredictable execution of the program and can provide opportunities for execution of unintended or malicious code.

The compilation systems for many languages and implementations can check to ensure that the list of actual parameters and any expected return match the declared set of formal parameters and return value (the *subprogram signature*) in both number and type. (In some cases, programmers should observe a set of conventions to ensure that this is true.) However, when the call is being made to an externally compiled subprogram, an object-code library, or a module compiled in a different language, the programmer must take additional steps to ensure a match between the expectations of the caller and the called subprogram.

6.2.5 Applicable language characteristics

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

- Languages that do not ensure automatically that the number and types of actual arguments are equal to the number and types of the formal parameters.
- Implementations that permit programs to call subprograms that have been externally compiled (without a means to check for a matching subprogram signature), subprograms in object code libraries, and subprograms compiled in other languages.

6.2.6 Avoiding the vulnerability or mitigating its effects

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

- Take advantage of any mechanism provided by the language to ensure that parameter signatures match.
- Avoid any language features that permit variable numbers of actual arguments without a method of enforcing a match for any instance of a subprogram call.
- Take advantage of any language or implementation feature that would guarantee matching the subprogram signature in linking to other languages or to separately compiled modules.
- Intensively review and subprogram calls where the match is not guaranteed by tooling.

6.2.7 Implications for standardization

Language specifiers could ensure that the signatures of subprograms match within a single compilation unit and could provide features for asserting and checking the match with externally compiled subprograms.

6.2.8 Bibliography

[None]

6.3 XYH Null Pointer Dereference

6.3.0 Status and history

OK: No one is assigned responsibility
2007-12-15, status updated, Jim Moore
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.3.1 Description of application vulnerability

A null-pointer dereference takes place when a pointer with a value of NULL is used as though it pointed to a valid memory area.

6.3.2 Cross reference

CWE:
467. Null Pointer Dereference

6.3.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.3.4 Mechanism of failure

A null-pointer dereference takes place when a pointer with a value of `NULL` is used as though it pointed to a valid memory area. Null-pointer dereferences often result in the failure of the process or in very rare circumstances and environments, code execution is possible.

6.3.5 Applicable language characteristics

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

- Languages that permit the use of pointers.
- Languages that allow the use of a `NULL` pointer.

6.3.6 Avoiding the vulnerability or mitigating its effects

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

- Before dereferencing a pointer, ensure it is not equal to `NULL`.

6.3.7 Implications for standardization

6.3.8 Bibliography

6.4 XYK Dangling Reference to Heap

6.4.0 Status and history

2008-02-14: minor wording changes and deletion of a complicated explanation that did not add much additional info, by Erhard Ploedereder
2007-12-14, reviewed and edited at OWGV meeting 7
2007-12-11, Edited by Erhard Ploedereder; general edits without any MISRA additions
2007-10-15, Decided at OWGV #6: We decide to write a new vulnerability, Pointer Arithmetic, RVG, for 17.1 thru 17.4. Don't do 17.5. We also want to create DCM to deal with dangling references to stack frames, 17.6. XYK deals with dangling pointers. Deal with MISRA 2004 rules 17.1, 17.2, 17.3, 17.4, 17.5, 17.6; JSF rule 175.
2007-10-01, Edited at OWGV #6
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.4.1 Description of application vulnerability

A dangling reference is a reference to an object whose lifetime has ended due to explicit deallocation or the stack frame in which the object resided has been freed due to exiting the dynamic scope. The memory for the object may be reused; therefore, any access through the dangling reference may affect an apparently arbitrary location of memory, corrupting data or code.

This description concerns the former case, dangling references to the heap. The description of dangling references to stack frames is DCM. In many languages references are called pointers; the issues are identical.

A notable special case of using a dangling reference is calling a deallocator, e.g., `free,` twice on the same memory address. Such a "Double Free" may corrupt internal data structures of the heap administration, leading to extremely surprising fault behaviour (such as infinite loops within the allocator, returning the same memory repeatedly as the result of distinct subsequent allocations, or deallocating memory legitimately allocated to another request since the first `free` call, to name but a few), or it may have no adverse effects at all.

Memory corruption through the use of a dangling reference is among the most difficult of errors to locate.

With sufficient knowledge about the heap management scheme (often provided by the OS or standard kernel), use of dangling references is an exploitable vulnerability, since the dangling reference provides an arbitrary view to read and modify valid data in the designated memory locations after freed memory has been re-allocated by subsequent allocations.

6.4.2 Cross reference

CWE:
 415. Double Free (Note that Double Free (415) is a special case of Use After Free (416))
 416. Use after Free
MISRA C 2004: 17.6

6.4.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.4.4 Mechanism of failure

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behaviour is undefined. Explicit deallocation of heap-allocated storage ends the lifetime of the object residing at this memory location (as does leaving the dynamic scope of a declared variable). The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime. Such pointers are called dangling references.

The use of dangling references to previously freed memory can have any number of adverse consequences — ranging from the corruption of valid data to the execution of arbitrary code, depending on the instantiation and timing of the deallocation causing all remaining copies of the reference to become dangling, of the system's reuse of the freed memory, and of the subsequent usage of a dangling reference.

Like memory leaks and errors due to double de-allocation, the use of dangling references has two common and sometimes overlapping causes: Error conditions and other exceptional circumstances; and confusion over which part of the program is responsible for freeing the memory. In one scenario, the memory in question is allocated validly to another pointer at some point after it has been freed. However, the original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed via this original pointer, it corrupts the validly re-used memory. This induces undefined behaviour in the affected program. If the newly allocated data happens to hold a class description, in C++ for example, various function pointers may be scattered within the heap data. If one of these function pointers is overwritten with an address of malicious code, execution of arbitrary code can be achieved.

6.4.5 Applicable language characteristics

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

- Languages that permit the use of pointers and that permit explicit deallocation by the user or provide for alternative means to reallocate memory still pointed to by some pointer value.

6.4.6 Avoiding the vulnerability or mitigating its effects

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

- Use a language or implementation that performs garbage collection rather than requiring the program to explicitly release allocated storage. In this case, the program must set all pointers/references to NULL when no longer needed (or else garbage collection will not collect the referenced memory). Alternatively use a language or implementation that provides for storage pools and performs deallocation upon leaving the scope of the pool.
- Use an implementation that checks whether a pointer is used that designates a memory location that has already been freed.
- Use a coding style that never permits deallocation.
- Ensure that each allocation is freed only once. After freeing a chunk of memory, set the pointer to NULL to ensure the pointer cannot be freed again. In complicated error conditions, be sure that clean-up routines respect the state of allocation properly. If the language is object-oriented, ensure that object destructors delete each chunk of memory only once. Ensuring that all pointers are set to NULL once memory they point to has been freed can be an effective strategy. The utilization of multiple or complex data structures may lower the usefulness of this strategy.
- Allocating and freeing memory in different modules and levels of abstraction burdens the programmer with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to programming defects such as double-free vulnerabilities, accessing freed memory, or dereferencing NULL pointers or uninitialized pointers. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

6.4.7 Implications for standardization

Implementations of the free function could tolerate multiple frees on the same reference/pointer or frees of memory that was never allocated.

An explicit deallocation should set the pointer to NULL to reduce the number of dangling references.

6.4.8 Bibliography

6.5 XYL Memory Leak

6.5.0 Status and history

PENDING
2008-01-14, Edited by Stephen Michell
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.5.1 Description of application vulnerability

[Note: Possibly separate item: Attempting to allocate storage and not checking if it is successful.]

The software does not sufficiently track and release allocated memory after it has been used, which slowly consumes remaining memory. This is often triggered by improper handling of malformed data or unexpectedly interrupted sessions. This can be used by attackers to generate denial-of-service attacks and can cause premature shutdown for safety-related systems..

6.5.2 Cross reference

CWE:
401. Memory Leak

6.5.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.5.4 Mechanism of failure

As a process or system runs, any memory taken from dynamic memory and not returned or reclaimed (by the runtime system or a garbage collector) becomes unusable, causing constantly more memory to be used with each iteration. Alternatively, memory claimed and partially returned can cause the heap to fragment, which will eventually result in an inability to take the necessary size storage. Either condition will result in a memory exhaustion exception, and program termination or a system crash.

If an attacker can determine the cause of the memory leak, an attacker may be able to cause the application to leak quickly and therefore cause the application to crash.

6.5.5 Applicable language characteristics

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

- All general-purpose languages have mechanisms to dynamically allocate memory and reclaim memory.
- Some languages, such as Ada, provide mechanisms to create specialized pools for the management of limited types and objects without corrupting a general heap.
- Some languages, such as Java, have the capability for garbage collection to collect dynamically allocated memory that is no longer reachable, but the reclamation computation is in general **hard** and may consume excessive system resources.
- Most languages provide a complete paradigm to manage space via global mechanisms without the need to resort to dynamic memory.

6.5.6 Avoiding the vulnerability or mitigating its effects

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

- Garbage collectors attempts to reclaim memory that will never be used by the application again. Some garbage collectors are part of the language while others are add-ons. Again, this is not a complete solution as it is not 100% effective, but it can significantly reduce the number of memory leaks.

- Allocating and freeing memory in different modules and levels of abstraction burdens the programmer and any static analysis tools with tracking the lifetime of that block of memory. This may cause confusion regarding when and if a block of memory has been allocated or freed, leading to memory leaks. To avoid these situations, it is recommended that memory be allocated and freed at the same level of abstraction, and ideally in the same code module.

- Storage pools are a specialized memory mechanism where all of the memory associated with a class of objects is allocated from a specific bounded region. When used with strong typing one can ensure a strong relationship between pointers and the space accessed such that storage exhaustion in one pool does not affect the code operating on other memory.
- Memory leaks can be eliminated by avoiding the use of dynamically allocated storage entirely, or by doing initial allocation exclusively and never allocating once the main execution commences.
- For safety-related systems and long running systems, the use of dynamic memory is almost always prohibited, or restricted to the initialization phase of execution.

6.5.7 Implications for standardization

Languages can provide syntax and semantics to guarantee program-wide that dynamic memory is not used (such as Ada's configuration pragmas).

Languages can document or can specify that implementations must document choices for dynamic memory management algorithms, to help designers decide on appropriate usage patterns and recovery techniques as necessary.

6.5.8 Bibliography

6.6 XYW Buffer Overflow in Stack

[Note: Recommend merging this with XZB.]

6.6.0 Status and history

PENDING2008-02-13, Edited by Derek Jones
2007-12-14, edited at OWGV meeting 7.
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.6.1 Description of application vulnerability

A buffer overflow occurs when a Standard library function is called to copy N bytes (or other units of storage) from one buffer to another and the amount being read/written is greater than is allocated for the source or destination buffer.

6.6.2 Cross reference

CWE:
[stack overflow is caused by deep nesting of function calls, so not applicable]

6.6.3 Categorization

See clause 5.?.
Group: Array Bounds

6.6.4 Mechanism of failure

Many languages and some third party libraries provide functions which efficiently copy one area of storage to another area of storage. Most of these libraries do not perform any checks to ensure that the copied from/to storage area is large enough to accommodate the amount of data being copied.

The arguments to these library functions include the addresses of the two storage areas and the number of bytes (or some other measure) to copy Passing the appropriate combination of incorrect start addresses or number of bytes to copy makes it is possible to read or write outside of the storage allocated to the source/destination area. When passed incorrect parameters the library function performs one or more unchecked array index accesses, as described in XYZ Unchecked Array Indexing.

6.6.5 Applicable language characteristics

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

- Languages that contain Standard library functions for performing bulk copying of storage areas.
- The same range of languages having the characteristics listed in XYZ Unchecked Array Indexing

6.6.6 Avoiding the vulnerability or mitigating its effects

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

- Only use library functions that perform checks on the arguments to ensure no buffer overrun can occur (perhaps by writing a wrapper for the Standard provided functions). Perform checks on the argument expressions prior to calling the Standard library function to ensure that no buffer overrun will occur.
- Use of static analysis to verify that the appropriate library functions are only called with arguments that do not result in a buffer overrun. 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.

6.6.7 Implications for standardization

6.6.8 Bibliography

[List some buffer bounds checking papers]

6.7 XZB Buffer Overflow in Heap

[Note: Recommend merging this with XYW.]

6.7.0 Status and history

PENDING
2008-02-13, Edited by Derek Jones
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.7.1 Description of application vulnerability

A heap overflow condition is a buffer overflow, where the buffer that can be overwritten is allocated in the heap portion of memory, generally meaning that the buffer was allocated using a routine such as the POSIX `malloc()` call.

6.7.2 Cross reference

CWE:

122. Heap Overflow

6.7.3 Categorization

See clause 5.?.
Group: Array Bounds

6.7.4 Mechanism of failure

Heap overflows are usually just as dangerous as stack overflows. Besides important user data, heap overflows can be used to overwrite function pointers that may be living in memory, pointing it to the attacker's code. Even in applications that do not explicitly use function pointers, the run-time will usually leave many in memory. For example, object methods in C++ are generally implemented using function pointers. Even in C programs, there is often a global offset table used by the underlying runtime.

Heap overflows generally lead to crashes. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop. Heap overflows can be used to execute arbitrary code, which is usually outside the scope of a program's implicit security policy. When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

6.7.5 Applicable language characteristics

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

- The size and bounds of arrays and their extents might be statically determinable or dynamic. Some languages provide both capabilities.
- Language implementations might or might not statically detect out of bound access and generate a compile-time diagnostic.
- At run-time the implementation might or might not detect the out of bounds access and provide a notification at run-time. The notification might be treatable by the program or it might not be.
- Accesses might violate the bounds of the entire array or violate the bounds of a particular extent. It is possible that the former is checked and detected by the implementation while the latter is not.
- The information needed to detect the violation might or might not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer might deprive the subroutine of information regarding the size of the array.)
- Some languages provide for whole array operations that may obviate the need to access individual elements.
- Some languages may automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds. (This may or may not match the programmer's intent.)

6.7.6 Avoiding the vulnerability or mitigating its effects

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

- Use a language or compiler that performs automatic bounds checking.
- Use an abstraction library to abstract away risky APIs, though not a complete solution.
- Canary style bounds checking, library changes which ensure the validity of chunk data and other such fixes are possible, but should not be relied upon.

- OS-level preventative functionality can be used, but is also not a complete solution.
- Protection to prevent overflows can be disabled in some languages to increase performance. This option should be used very carefully.

6.7.7 Implications for standardization

6.7.8 Bibliography

6.8 BQF Unspecified Behaviour

6.8.0 Status and History

2008-02-12, Revised by Derek Jones
2007-12-12: Considered at OWGV meeting 7: In general, it's not possible to completely avoid unspecified behaviour. The point is to code so that the behaviour of the program is indifferent to the lack of specification. In addition, Derek should propose additional text for Clause 5 that explains that different languages use the terms "unspecified", "undefined", and "implementation-defined" in different ways and may have additional relevant terms of their own. Also, 5.1.1 should clarify that the existence of unspecified behaviour is not necessarily a defect, or a failure of the language specification. N0078 may be helpful.
2007-10-15, Jim Moore added notes from OWGV Meeting #6: "So-called portability issues should be confined to EWF, BQF, and FAB. The descriptions should deal with MISRA 2004 rules 1.2, 3.1, 3.2, 3.3, 3.4 and 4.a; and JSF C++ rules 210, 211, 212, 214. Also discuss the role of pragmas and assertions."
2007-07-18, Edited by Jim Moore
2007-06-30, Created by Derek M. Jones

6.8.1 Description of application vulnerability

The external behavior of a program, whose source code contains one or more instances of constructs having unspecified behavior, when the source code is recompiled or relinked.

6.8.2 Cross reference

Ada: Clause 1.1.3 Conformity of an Implementation with the Standard; Clause 3.4.4 unspecified behavior
C: Clause 3.4.4 unspecified behavior
C++: Clause 1.3.13 unspecified behavior
Fortran: Clause 1.5 Conformance (Fortran uses the term 'processor dependent')
Also see guideline recommendations: EWF-undefined-behavior and FAB-implementation-defined-behavior.

6.8.3 Categorization

See clause 5.1.1.

6.8.4 Mechanism of failure

Language specifications do not always uniquely define the behavior of a construct. When an instance of a construct that is not uniquely defined is encountered (this might be at any of compile, link, or run time) implementations are permitted to choose from the set of behaviors allowed by the language specification. The term 'unspecified behavior' is sometimes applied to such behaviors, (language specific guidelines need to analyse and document the terms used by their respective language).

A developer may use a construct in a way that depends on a subset of the possible behaviors occurring. The behavior of a program containing such a usage is dependent on the translator used to build it always selecting the 'expected' behavior.

6.8.5 Interrupting the Failure Mechanism

Many language constructs may have unspecified behavior and unconditionally recommending against any use of these constructs may be impractical. For instance, in many languages the order of evaluation of the operands appearing on the left- and right-hand side of an assignment statement is unspecified, but in most cases the set of possible behaviors always produces the same result.

The appearance of unspecified behavior in a language specification is a recognition by the designers that in some cases flexibility is needed by software developers and provides a worthwhile benefit for language translators; this usage is not a defect in the language.

The important characteristic is not the internal behavior exhibited by a construct (e.g., the sequence of machine code generated by a translator) but its external behavior (i.e., the one visible to a user of a program). If the set of possible unspecified behaviors permitted for a specific use of a construct all produce the same external effect when the program containing them is executed, then rebuilding the program cannot result in a change of behavior for that specific usage of the construct.

For instance, while the following assignment statement contains unspecified behavior in many languages (I.e., it is possible to evaluate either A or B operand first, followed by the other operand):

```
A = B;
```

in most cases the order in which `A` and `B` are evaluated does not effect the external behavior of a program containing this statement.

6.8.6 Assumed variations among languages

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

- languages whose specification allows a finite set of more than one behaviors for how a translator handles some construct, where two or more of the behaviors can result in differences in external program behavior.

6.8.7 Avoiding the vulnerability or mitigating its effects

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

- Ensuring that a specific use of a construct having unspecified behavior produces a result that is the same, for that specific use, for all of possible behaviors permitted by the language specification.

When developing coding guidelines for a specific language all constructs that have unspecified behavior shall be documented and for each construct the situations where the set of possible behaviors can vary shall be enumerated.

6.8.8 Bibliography

6.9 EWF Undefined Behaviour

6.9.0 Status and history

2008-02-11, Revised by Derek Jones
2007-12-12, Considered at OWGV meeting 7: Clarify that different languages use different terminology. Also consider Tom Plum's paper N0104.
2007-10-15, Jim Moore added notes from OWGV Meeting #6: "So-called portability issues should be confined to EWF, BQF, and FAB. The descriptions should deal with MISRA 2004 rules 1.2, 3.1, 3.2, 3.3, 3.4 and 4.a; and JSF C++ rules 210, 211, 212, 214. Also discuss the role of pragmas and assertions."
2007-07-19, Edited by Jim Moore

2007-06-30, Created by Derek M. Jones

6.9.1 Description of application vulnerability

The external behaviour of a program containing an instance of a construct having undefined behaviour, as defined by the language specification, is not predictable.

6.9.2 Cross reference

Ada: Clause 1.1.5 Classification of Errors (the term “bounded error” is used in a way that is comparable with undefined behavior).
C: Clause 3.4.3 undefined behaviour
C++: Clause 1.3.12 undefined behaviour
Fortran: ??? [The terms 'undefined behavior', 'illegal', 'non-conforming', and 'non-standard' do not appear in the Fortran Standard. 'Undefined' is used in the context of a variable having an undefined value. Does Fortran have any concept of undefined behavior? Need to talk to Fortran people.]
Also see guideline recommendations: BQF-071212-unspecified-behavior and FAB-implementation-defined-behavior.

6.9.3 Categorization

See clause 5.1.3.

6.9.4 Mechanism of failure

Language specifications may categorize the behavior of a language construct as undefined rather than as a semantic violation (I.e., an erroneous use of the language) because of the potentially high implementation cost of detecting and diagnosing all occurrences of it. In this case no specific behaviour is required and the translator or runtime system is at liberty to do anything it pleases (which may include issuing a diagnostic).

The behavior of a program built from successfully translated source code containing an instance of a construct having undefined behavior is not predictable.

6.9.5 Applicable language characteristics

This vulnerability can be avoided by not using any construct that has undefined behavior or by using the construct in a way that guarantees that the domain of its operation (e.g., the value of the right operand of a division operator is never zero) does not fall into undefined behaviour.

6.9.6 Assumed variations among languages

- Languages vary in the extent to which they specify the use of a particular construct to be a violation of the specification or undefined behavior. They also vary on whether the behavior is said to occur during translation, link-time, or during program execution.

6.9.7 Avoiding the vulnerability or mitigating its effects

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

- Ensuring that the language construct is not used.
- Ensuring that a use of a construct having undefined behaviour does not operate within the domain in which the behaviour is undefined. When it is not possible to completely verify the domain of operation during translation a run-time check may need to be performed.

When developing coding guidelines for a specific language all constructs that have undefined behavior shall be documented. The items on this list might be classified by the extent to which the behavior is likely to have some critical impact on the external behavior of a program (the criticality may vary between different implementations, e.g., whether conversion between object and function pointers has well defined behavior).

6.9.8 Bibliography

6.10 XYY Wrap-around Error

6.10.0 Status and history

PENDING
2008-01-12, Edited by Dan Nagle
2007-10-01, Edited at OWGV #6
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.10.1 Description of application vulnerability

Wrap around errors occur whenever a value is incremented past the maximum value for its type and therefore "wraps around" to a very small, negative, or undefined value. Shift operations may produce a similar error.

[Generalize this to deal with the undefined aspects of C shift operators.]

6.10.2 Cross reference

CWE:
128. Wrap-around Error
MISRA C 2004: 12.8

6.10.3 Categorization

See clause 5.?.
Group: Arithmetic

6.10.4 Mechanism of failure

Due to how arithmetic is performed by computers, if a primitive is incremented past the maximum value possible for its type, the system may fail to provide an indication to the program, and therefore increment each bit as if it still had extra space. Because of how negative numbers are represented in binary, primitives interpreted as signed may "wrap" to very large negative values.

Shift operations may also produce values that cannot be easily predicted as a result of the different representations of negative integers on various hardware, and, when treating signed quantities, of the differences in behavior between logical shifts and arithmetic shifts (the particular effect filling with the sign bit).

Wrap-around errors generally lead to undefined behavior and infinite loops, and therefore crashes. If the value in question is important to data (as opposed to flow), data corruption will occur. If the wrap around results in other conditions such as buffer overflows, further memory corruption may occur. A wrap-around can sometimes trigger buffer overflows which can be used to execute arbitrary code.

6.10.5 Applicable language characteristics

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

- Some languages do not trigger an exception condition when a wrap-around error occurs.

- Some languages do not fully specify the distinction between arithmetic and logical shifts

6.10.6 Avoiding the vulnerability or mitigating its effects

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

- The choice could be made to use a language that is not susceptible to these issues.
- Provide clear upper and lower bounds on the scale of any protocols designed.
- Place sanity checks on all incremented variables to ensure that they remain within reasonable bounds.
- Analyze the software using static analysis.
- Use one set of variables for quantities involved in shift operations, and another for quantities involved in arithmetic operations. Check values when assignments are made between the two sets.

6.10.7 Implications for standardization

6.10.8 Bibliography

6.11 XYQ Dead and Deactivated Code

[Note: ***It's possible that this should be retitled as "Dead Code". There should be a separate item for code that executes with no effect. It is indicative of a programming error.*****]**

[Note: The vulnerability as currently written – 2008-01-02 – also talks about dead code that is inadvertently or unexpectedly executed. Whilst this is clearly closely related to dead code, consideration might be given to making it a new vulnerability.]

6.11.0 Status and history

2008-01-02, Updated by Clive Pygott
2007-12-13, OWGV Meeting 7 considered the draft: This should be merged with the proposal regarding dead code in N0108. Also the decision made at meeting 6 should be implemented.
2007-12-13, OWGV Meeting 7 renamed this from "Expression Issues" to "Dead and Deactivated Code"
2007-10-15, OWG Meeting 6 decided: " XYQ concerns code that cannot be reached. That is somewhat different than code that executes with no result. The latter is a symptom of poor quality code but may not be a vulnerability. We should introduce a new item, KOA, for code that executes with no result because it is a symptom of misunderstanding during development or maintenance. (Note that this is similar to unused variables.) We probably want to exclude cases that are obvious, such as a null statement, because they are obviously intended. It might be appropriate to require justification of why this has been done. These may turn out to be very specific to each language. The rule needs to be generalized."
Also: Deal with reachability of statement – MISRA rules 14.1 and 2.4. JSF rule 127.
2007-10-01, Edited at OWGV Meeting #6
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-19, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.11.1 Description of application vulnerability

Dead and Deactivated code (the distinction is addressed in 6.11.4) is code that exists in the executable, but which can never be executed, either because there is no call path that leads to it (e.g. a function that is never called), or the path is semantically infeasible (e.g. its execution depends on the state of a conditional that can never be achieved).

Dead and Deactivated code is undesirable because it indicates the possibility of a coding error and because it may provide a "jump" target for an intrusion.

Also covered in this vulnerability is code which is believed to be dead, but which is inadvertently executed.

6.11.2 Cross reference

BVQ: Unspecified functionality. Unspecified functionality is unnecessary code that exists in the binary which may be executed. Dead and deactivated code is unnecessary code that exists in the binary but can never be executed.

CWE:
570. Expression is Always True
571. Expression is Always False
MISRA C 2004: 14.1, 2.4
MISRA C++ 0-1-1 0-1-2 0-1-9 0-1-10
DO178B/C

6.11.3 Categorization

6.11.4 Mechanism of failure

DO-178B defines Dead and Deactivated code as:

Dead code – Executable object code (or data) which... cannot be executed (code) or used (data) in an operational configuration of the target computer environment and is not traceable to a system or software requirement.

Deactivated code – Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options.]

Dead code is code that exists in an application, but which can never be executed, either because there is no call path to the code (e.g. a function that is never called) or because the execution path to the code is semantically infeasible, e.g. in

```
if(true) A;  else B;
```

`B` is dead code, as only `A` can ever be executed.

The presence of dead code is not in itself an error, but begs the question why is it there? Is its presence an indication that the developer believed it to be necessary, but some error means it will never be executed? Or is there a legitimate reason for its presence, for example:

- as defensive code, only executed as the result of a hardware failure
- as part of a library not required in this application
- as diagnostic code not executed in the operational environment

Such code may be referred to as “deactivated”. That is, dead code that is there by intent.

There is a secondary consideration for dead code in languages that permit overloading of functions etc. and use complex name resolution strategies. The developer may believe that some code is not going to be used (deactivated), but its existence in the program means that it appears in the namespace, and may be selected as the best match for some use that was intended to be of an overloading function. That is, although the developer believes it is never going to be used, in practice it is used in preference to the intended function.

6.11.5 Applicable language characteristics

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

- code that exists in the executable that can never be executed
- code that exists in the executable that was not expected to be executed, but is.

6.11.6 Avoiding the vulnerability or mitigating its effects

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

- as a first resort, the developer should endeavour to remove, as far as is practical, dead code from an application.
- where code is dead because a conditional always evaluates to the same value, this could be indicative of an earlier bug and additional testing may be needed to ascertain why the same value is occurring
- notwithstanding the above, the developer should identify any dead code in the application, and provide a justification (if only to themselves) as to why it is there
- the developer should also ensure that any code that was expected to be unused is actually recognised as dead

6.11.7 Implications for standardization

6.11.8 Bibliography

Hatton 2003
MISRA C 2004

6.12 XYR Unused Variable

6.12.0 Status and history

2008-02-14 a serious rewrite to separate unused declarations from dead stores; the previous version merged their causes, effects and remedies in incorrect ways; by Erhard Ploedereder
2007-12-14, revise to deal with this comment: " also closely related is reassigning a value to a variable without evaluating it" in 6.12.5.
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-19, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.12.1 Description of application vulnerability

A variable's value is assigned but never used, making it a dead store. As a variant, a variable is declared but neither read nor written to in the program, making it an unused variable.

6.12.2 Cross reference

CWE:
563. Unused Variable

6.12.3 Categorization

See clause 5.?.

6.12.4 Mechanism of failure

A variable is declared, but never used. It is likely that the variable is simply vestigial, but it is also possible that the unused variable points out a bug.

A variable is assigned a value but this value is never used thereafter. The assignment is then generally referred to as a dead store. Note that this may be acceptable if the variable is a volatile variable, for which the assignment of a value triggers some external event.

A dead store is indicative of sloppy programming or of a design or coding bug: either the use of the value was forgotten (almost certainly bug) or the assignment was done even though it was not needed (sloppiness).

An unused variable or a dead store is very unlikely to be the cause of a vulnerability. However, since compilers diagnose unused variables routinely and dead stores occasionally, their presence is often an indication that compiler warnings are either suppressed or are being ignored by programmers – a vulnerability in its own right. This observation does not hold for automatically generated code, where it is commonplace to find unused variables and dead stores, introduced to keep the generation process simple and uniform.

6.12.5 Applicable language characteristics

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

- Dead stores are possible only in languages that provide assignment. (Pure functional languages do not have this issue.)

- Unused variables (in the technical sense above) are possible only in languages that provide variable declarations. Languages, in which instead the first assignment introduces the variable, the identical issue of no further uses maps onto the problem of dead stores.

6.12.6 Avoiding the vulnerability or mitigating its effects

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

- Enable detection of unused variables and dead stores in the compiler. The default setting may be to suppress these warnings.

6.12.7 Implications for standardization

- Consider mandatory diagnostics for unused variables.

6.12.8 Bibliography

6.13 XYX Boundary Beginning Violation

[Note: Perhaps this should be subsumed by XYZ.]

[Note: Added this recommendation to XYZ Unchecked Array Indexing]

6.13.0 Status and history

2008-02-13, Edited by Derek Jones
2007-12-14, edited at OWGV meeting 7
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.13.1 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.

6.13.2 Cross reference

CWE:
124. Boundary Beginning Violation ("buffer underwrite")
129. Unchecked Array Indexing

6.13.3 Categorization

See clause 5.?.
Group: Array Bounds

6.13.4 Mechanism of failure

There are two kinds of failures (jn both 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 uninitialised 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 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.

6.13.5 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.

6.13.6 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 which significantly reduces the runtime overhead (because runtime checks are only made when it cannot be shown statically that no bound violations can occur).

6.13.7 Implications for standardization

6.13.8 Bibliography

6.14 XZI Sign Extension Error

6.14.0 Status and history

REVISE: Tom Plum
2008-01-16, Edited by Plum [and suggest it be merged into FLC]
2007-12-14, considered at OWGV meeting 7. Some issues are noted below.
2007-08-05, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.14.1 Description of application vulnerability

If one extends a signed number incorrectly, if negative numbers are used, an incorrect extension may result.

[Consider the two issues listed immediately below:]
[Note: combining XYE [subsumed by FLC]
XYF [subsumed by FLC],
XYY [just revised by Dan],
XZI as "integer arithmetic" was suggested.] I agree; merge it into FLC.
[Note: Should "divide by zero" be added?] I recommend a separate new topic.

6.14.2 Cross reference

CWE:
194. Sign Extension Error

6.14.3 Categorization

See clause 5.?.
Group: Arithmetic

6.14.4 Mechanism of failure

Converting a signed shorter data type to a larger data type or pointer can cause errors due to the extension of the sign bit. A negative data element that is extended with an unsigned extension algorithm will produce an incorrect result. For instance, this can occur when a signed character is converted to a short or a signed integer is converted to a long. Sign extension errors can lead to buffer overflows and other memory based problems. This can occur unexpectedly when moving software designed and tested on a 32 bit architecture to a 64 bit architecture computer.

[To understand the topic better, I consulted the original CWE

The following example is provided:

```
struct fakeint { short f0; short zeros; };
```

```
    struct fakeint strange;
    struct fakeint strange2;
    strange.f0=-240;
    strange2.f0=240;
    strange2.zeros=0;
    strange.zeros=0;
    printf("%d %d\n",strange.f0,strange);
    printf("%d %d\n",strange2.f0,strange2);
```

Maybe I just need more coffee, but this looks wrong. Negative 240 will assign just fine to a short integer which C90 and C99 require to be at least 16 bits. If “short” is changed to “unsigned char”, then the example illustrates the text. (If the C/C++ folks agree with that, I’ll draft a suggestion to CWE.)

6.14.5 Applicable language characteristics

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

- Languages may be strongly or weakly typed. Strongly typed languages do a strict enforcement of type rules since all types are known at compile time.
- Some languages allow implicit type conversion. Others require explicit type conversion.

6.14.6 Avoiding the vulnerability or mitigating its effects

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

- Use a sign extension library, standard function, or appropriate language-specific coding methods to extend signed numbers.
- Use static analysis tools to help locate situations in which unintended conversions could affect numerical values.

6.14.7 Implications for standardization

6.14.8 Bibliography

6.15 XZH Off-by-one Error

6.15.0 Status and history

2007-12-28, Edited by Stephen Michell
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-19, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.15.1 Description of application vulnerability

A product uses an incorrect maximum or minimum value that is 1 more or 1 less than the correct value. This usually arises from one of a number of situations where the bounds as understood by the developer differ from the design, such as;

- confusion between the need for “<” and “<=” or “>” and “>=” in a test
- confusion as to the sentinels (start point and end point) for an algorithm, such as beginning an algorithm at 1 when the underlying structure is indexed from 0, beginning an algorithm at 0 when the underlying structure is indexed from 1 (or some other start point) or using the length or a structure as the count mechanism instead of the sentinel values

These issues arise from mistakes in mapping the design into a particular language, in moving between languages (such as between C-based languages where all arrays start at 0 and Pascal-based languages where all arrays start at 1 or Ada where all bounds are specifiable), and when exchanging data between languages with different default array sentinel values.

The issue also can arise in more complex algorithms where relationships exist between components, and the existence of a sentinel value changes the conditions of the test.

The existence of this possible flaw can also be a serious security hole as it can permit someone to surreptitiously provide an unused location (such as 0 or the last element) that can be used for undocumented features or hidden channels).

6.15.2 Cross reference

CWE:
193. Off-by-one Error

6.15.3 Categorization

See clause 5.?.

6.15.4 Mechanism of failure

An off-by-one error could lead to.

- an out-of bounds access to an array (buffer overflow),
- an incomplete comparisons and calculation mistakes,
- a read from the wrong memory location, or
- an incorrect conditional.

Such incorrect accesses can cause calculation errors or references to illegal locations, resulting in potentially unbounded behaviour.

Off-by-one errors are not exploited as often in attacks because they are difficult to identify and exploit externally, but the calculation errors and boundary-condition errors can be severe.

6.15.5 Applicable language characteristics

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

- Many languages have mechanisms to help avoid mistakes in bounding array accesses correctly, e.g. methods to obtain the actual bounds of an array.
- Some languages provide mechanisms such as iterators or whole array operations to access whole arrays without explicitly naming the sentinel values
- Some languages provide mechanisms to reference key array elements and properties, such as first, last, next, previous and length that help programmers avoid off-by-one errors.
- Most languages provide named constants for sentinals (including the lower bound) whose use in iterators and bounds checks dramatically reduces errors of this type.
- Some languages start all arrays at the 0th element, some at the 1st element and some permit programmer-specified bounds.
- Some language provide iterators that work on the specific structure that owns the iterator.

6.15.6 Avoiding the vulnerability or mitigating its effects

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

- Off-by-one errors are a common bug that is also a code quality issue. As with most quality issues, a systematic development process, use of development/analysis tools and thorough testing are all common ways of preventing errors, and in this case, off-by-one errors.

- Where references are being made to structure indices and the languages provide ways to specify the whole structure or the starting and ending indices explicitly (eg Ada provides xxx'First and xxx'Last for each dimension), these should be used always. Where the language doesn't provide these, constants can be declared and used in preference to numeric literals.

```
const int str_first     =   0;
const int str_last     =  99;
const int str_length = 100;
char str[str_last];
for (i = str_first; i <= str_last; i++) {... do stuff}
```

- Coding standards can be written such that either the sentinal values or the length of all arrays is used. Ideally length should be a calculated function of the indices to avoid interpretation errors.

6.15.7 Implications for standardization

Languages can provide standard ways to access all elements in indexed structures without the need for numeric literals, as well as tests to ensure that algorithms cover the declared ranges of whole structures.

6.15.8 Bibliography

6.16 XYZ Unchecked Array Indexing

[Note: Suggest merging with XYX-Boundary Beginning Violation which also deals with indexing into an array. The name of the merged vulnerability to be "Unchecked Array Indexing"..]

6.16.0 Status and history

2008-02-13, Edited by Derek Jones
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

6.16.1 Description of application vulnerability

Unchecked array indexing occurs when an unchecked value is used as an index into a buffer.

6.16.2 Cross reference

CWE:
129. Unchecked Array Indexing

6.16.3 Categorization

See clause 5.?.
Group: Array Bounds

6.16.4 Mechanism of failure

A single fault could allow both an overflow and underflow of the array index. An index overflow exploit might use buffer overflow techniques, but this can often be exploited without having to provide "large inputs." Array index overflows can also trigger out-of-bounds read operations, or operations on the wrong objects; i.e., "buffer overflows" are not always the result.

Unchecked array indexing, depending on its instantiation, can be responsible for any number of related issues. Most prominent of these possible flaws is the buffer overflow condition. Due to this fact, consequences range from denial of service, and data corruption, to full blown arbitrary code execution. The most common condition situation leading to unchecked array indexing is the use of loop index variables as buffer indexes. If the end condition for the loop is subject to a flaw, the index can grow or shrink unbounded, therefore causing a buffer overflow or underflow. Another common situation leading to this condition is the use of a function's return value, or the resulting value of a calculation directly as an index in to a buffer.

Unchecked array indexing will very likely result in the corruption of relevant memory and perhaps instructions, leading to a crash, if the values are outside of the valid memory area. If the memory corrupted is data, rather than instructions, the system will continue to function with improper values. If the memory corrupted memory can be effectively controlled, it may be possible to execute arbitrary code, as with a standard buffer overflow.

6.16.5 Applicable language characteristics

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

- The size and bounds of arrays and their extents might be statically determinable or dynamic. Some languages provide both capabilities.
- Language implementations might or might not statically detect out of bound access and generate a compile-time diagnostic.

- At run-time the implementation might or might not detect the out of bounds access and provide a notification at run-time. The notification might be treatable by the program or it might not be.
- Accesses might violate the bounds of the entire array or violate the bounds of a particular extent. It is possible that the former is checked and detected by the implementation while the latter is not.
- The information needed to detect the violation might or might not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer might deprive the subroutine of information regarding the size of the array.)
- Some languages provide for whole array operations that may obviate the need to access individual elements.
- Some languages may automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds. (This may or may not match the programmer's intent.)

6.16.6 Avoiding the vulnerability or mitigating its effects

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

- Include sanity checks to ensure the validity of any values used as index variables. In loops, use greater-than-or-equal-to, or less-than-or-equal-to, as opposed to simply greater-than, or less-than compare statements.
- The choice could be made to use a language that is not susceptible to these issues

6.16.7 Implications for standardization

6.16.8 Bibliography

6.17 AMV Type-breaking reinterpretation of data

[For easy reference by reviewers, I have quoted the relevant JSF rules below:
[AV Rule 153 (MISRA Rule 110, Revised) Unions shall not be used.
[AV Rule 183 Every possible measure should be taken to avoid type casting.]

[For easy reference by reviewers, I have paraphrased the relevant MISRA 2004 rules below:
[Rule 18.2: Do not assign an object to an overlapping object.
[Rule 18.3: Do not reuse an area of memory for an unrelated purpose.
[Rule 18.4: Do not use unions.]

6.17.0 Status and history

2007-12-17: Jim Moore: Revised to implement comments from OWGV meeting 7. Changes determined by OWGV were simply accepted. The changes that I composed are marked with Track Changes.
2007-12-12: reviewed by OWGV meeting 7 with changes marked. Names was changed from "overlapping memory". Also rewrite to avoid the term "aliasing".
2007-12-05: revised by Moore
2007-11-26: Reformatted by Benito
2007-11-24: drafted by Moore
2007-10-15: OWGV meeting 6 decided: Write a new description, AMV. Overlapping or reuse of memory provides aliasing effects that are extremely difficult to analyze. Attempt to use alternative techiques when possible. If essential to the function of the program, document it clearly and use the clearest possible approach to implementing the function. (This includes C unions, Fortran common.) Discuss the difference between discriminating and non-discriminating unions. Discuss the possibility of computing the discriminator from the indiscriminate part of the union. Deal with unchecked conversion (as in Ada) and reinterpret casting (in C++). Deal with MISRA 2004 rules 18.2, 18.3, 18.4; JSF rules 153, 183.

6.17.1 Description of application vulnerability

In most cases, objects in programs are assigned locations in processor storage to hold their value. If the same storage space is assigned to more than one object—either statically or temporarily—then a change in the value of one object will have an effect on the value of the other. Furthermore, if the representation of the value of an object is reinterpreted as being the representation of the value of an object with a different type, unexpected results may occur.

6.17.2 Cross reference

CWE:
MISRA 2004: 18.2, 18.3, 18.4
JSF: 153, 183

6.17.3 Categorization

See clause 5.?.

6.17.4 Mechanism of failure

Sometimes there is a legitimate need for computer codes to place different interpretations upon the same stored representation of data. The most fundamental example is a program loader that treats a binary image of program as data by loading it, and then treats it as a program by invoking it. Most programming languages permit type-breaking reinterpretation of data, however, some offer safer alternatives for commonly encountered situations.

Type-breaking reinterpretation of representation presents obstacles to human understanding of the code, the ability of tools to perform effective static analysis, and the ability of code optimizers to do their job.

Examples include:

- Providing alternative mappings of objects into blocks of storage, performed either statically (e.g. Fortran `common`) or dynamically (e.g. pointers).
- Union types, particularly unions that do not have a discriminant stored as part of the data structure.
- Operations that permit a stored value to be interpreted as a different type (e.g. treating the representation of a character as an integer).

In all of these cases, the mechanism of failure is simple. Accessing the value of an object produces an unanticipated result.

A related problem, the aliasing of parameters, occurs in languages that permit call by reference because supposedly distinct parameters might refer to the same storage area, or a parameter and a non-local object might refer to the same storage area. That vulnerability is described in CSJ.

6.17.5 Applicable language characteristics

This vulnerability description applies to most procedural programming language.

6.17.6 Avoiding the vulnerability or mitigating its effects

This vulnerability cannot be completely avoided because some software codes necessarily view their stored data in alternative manners. However, these situations are unusual. Programmers should avoid reinterpretation performed as a matter of convenience; for example, using an integer pointer to manipulate character string data should be avoided. When type-breaking reinterpretation is necessary, it should be carefully documented in the code.

When using union types it is preferable to use discriminated unions. This is a form of a union where a stored value indicates which interpretation is to be placed upon the data. Some languages (e.g. variant records in Ada) enforce the view of data indicated by the value of the discriminant. If the language does not enforce the interpretation (e.g. equivalence in Fortran and union in C and C++), then the code should implement an explicit discriminant and check its value before accessing the data in the union, or use some other mechanism to ensure that correct interpretation is placed upon the data value.

Operations that reinterpret the same stored value as representing a different type should be avoided. The simplest case occurs in languages where the conversion is easily recognized as such. For example, the name of Ada's **Unchecked_Conversion** function explicitly warns of the problem. A much more difficult situation occurs when pointers are used for the same purpose. Some languages perform type-checking of pointers and place restrictions on the ability of pointers to access arbitrary locations in storage. Others permit the free use of pointers. In such cases, code must be carefully reviewed in a search for unintended reinterpretation of stored values. Therefore it is important to explicitly comment upon *intended* reinterpretations.

Static analysis tools may be helpful in locating situations where unintended reinterpretation occurs. On the other hand, the presence of reintepretation greatly complicates static analysis for other problems, so it may be appropriate to segregate intended reinterpretation operations into distinct subprograms.

6.17.7 Implications for standardization

Because the ability to perform reinterpretation is necessary, but the need for it is rare, programming language designers might consider putting caution labels on operations that permit reinterpretation. For example, the operation in Ada that permits unconstrained reinterpretation is called "**Unchecked_Conversion**".

Because of the difficulties with undiscriminated unions, programming language designers might consider offering union types that include distinct discriminants with appropriate enforcement of access to objects.

6.17.8 Bibliography

[1] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[2] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

6.18 BRS Leveraging human experience (was Maintability)

6.18.0 Status and history

2008-01-21, edited by Plum
2007-12-12, edited at OWGV meeting 7
2007-11-26, reformatted by Benito
2007-11-22, edited by Plum
2007-10: Assigned by OWG meeting 6: Write a new description, BRS, that says that guidelines for coding constructs should consider the capabilities of the review and maintenance audience as well as the writing audience, and that features that correlate with high error rates should be discouraged. Write another description, NYY, for self-modifying code that includes Java dynamic class libraries and DLLs. MISRA 12.10

6.18.1 Description of application vulnerability

Methodologies for developing critical software components (whether safety-critical, security-critical, mission-critical, etc.) will often require the participation of subject-matter experts, hardware engineers, human-factors engineers, safety officers, etc., in code reviews. This is one reason to develop and adopt guidelines to prohibit the use of language features which have been found to be obscure or misleading to this general audience.

Furthermore, the programmers who eventually maintain the system will often have less programming experience than the original developers; this provides another reason for the same type of prohibition. And in any event, consistency in coding is desirable for its own sake, so empirical support is not necessarily needed for all guidelines.

Experienced developers may determine that certain language features or programming constructs have a strong correlation with high error rates. Therefore, guidelines for developing critical software will generally discourage the use of such features or constructs.

6.18.2 Cross reference

MISRA C: 12.10
CERT/CC guidelines: MSC 05-A, 30-C, 31-C.

6.18.3 Categorization

[tbd].

6.18.4 Mechanism of failure

[tbd]

6.18.5 Applicable language characteristics

This vulnerability description is intended to be applicable to any languages.

6.18.6 Avoiding the vulnerability or mitigating its effects

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

- Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from CERT/CC, Hatton, or MISRA C.

6.18.7 Implications for standardization

[tbd]

6.18.8 Bibliography

Hatton 17: Use of obscure language features

6.19 CLL Switch statements and static analysis (was enumerable types)

6.19.0 Status and history

2008-01-25, edited by Plum
2007-12-12, edited at OWGV meeting 7
2007-11-26, reformatted by Benito
2007-11-22, edited by Plum
2007-10: OWGV meeting 6: Write a new description, CLL. Using an enumerable type is a good thing. One wants the case analysis to cover all of the cases. One often wants to avoid falling through to subsequent cases. Adding a default option defeats static analysis. Providing labels marking the programmer's intentions about falling through can be an aid to static analysis.6.x.1 Description of application vulnerability

6.19.1 Description of application vulnerability

When using a switch statement, it is important to make sure that all possible cases are, in fact, dealt with. One way to accomplish this is to switch on a variable of an enumerated type. In this case, it is preferable to omit the default case, because the static analysis is simplified and because maintainers can better understand the intent of the original programmer. When one must switch on some other form of type, it is necessary to have a default case, preferably to be regarded as an error condition. [Note from Tom: sounds exactly right to me]

In the `switch` statement of some languages, control can "flow-through" from one case into another case; this can result in execution of un-intended code. In general, it is preferable to avoid flowing through. (Multiple labels on one case are usually acceptable, even desirable.) In cases where it is necessary, providing comments marking the programmer's intentions about falling through can be an aid to static analysis and the understanding of maintainers.

In most languages, oversights during implementation can result in the omission of significant cases that should have been explicitly handled in a `switch` statement. Sometimes static analysis can assist with verifying that each significant case in the requirements is implemented in the corresponding **switch** statement; but if this assistance is employed, then a `default` case can diminish the effectiveness, since the tool cannot tell whether the omitted case was or was not intended for the **default** treatment **[Note from Tom: Jim's new text, two paragraphs earlier, already covered this adequately.]** Using an enumerable type for the switch variable can facilitate the assistance from static analysis, since the list of significant cases is more apparent from the declaration of the enumeration.

6.19.2 Cross reference

Hatton 14: Switch statements
MISRA C: 15.2, 15.3, add-in 14.8, 15.1, 15.4, 15.5
CERT/CC guidelines: MSC01-A

6.19.3 Categorization

[tbd].

6.19.4 Mechanism of failure

[tbd]

6.19.5 Applicable language characteristics

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

- Selection among alternative control-flow (`switch` statement or equivalent);
- Ability to flow-through from one case to another within a `switch`;
- Enumeration variables.

6.19.6 Avoiding the vulnerability or mitigating its effects

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

- Adopt appropriate programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from CERT/CC, Hatton, or MISRA C.
- Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, etc.

6.19.7 Implications for standardization

[Note: Perhaps languages could check for "completeness" of the switch variable and mutual exclusion of the cases.]

6.19.8 Bibliography

6.20 EOJ Demarcation of control flow (was Surprise in Control Flow)

6.20.0 Status and history

2008-01-22, edited by Plum
2007-12-12, edited at OWGV meeting 7
2007-11-26, reformatted by Benito
2007-11-22, edited by Plum

6.20.1 Description of application vulnerability

Some programming languages explicitly mark the end of an `if` statement or a loop, whereas other languages mark only the end of a block of statements. Languages of the latter category are prone to oversights by the programmer, causing unintended sequences of control flow.

6.20.2 Cross reference

Hatton 18: Control flow – if structure
MISRA C: 14.9, 14.10
JSF AV rules 59, 192

6.20.3 Categorization

[tbd].

6.20.4 Mechanism of failure

[tbd]

6.20.5 Applicable language characteristics

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

- Loops and conditional statements are not explicitly terminated by an “end” construct.

6.20.6 Avoiding the vulnerability or mitigating its effects

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

- Adopt a convention for marking the closing of a construct that can be checked by a tool, to ensure that program structure is apparent.
- Adopt programming guidelines (preferably augmented by static analysis analysis). For example, consider the rules itemized above from Hatton, JSF AV, or MISRA C.
- Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, etc.
- Pretty-printers and syntax-aware editors may be helpful in finding such problems, but sometimes disguise them.

6.20.7 Implications for standardization

Specifiers of languages might consider explicit termination of loops and conditional statements. Specifiers might consider features to terminate named loops and conditionals and determine if the structure as named matches the structure as inferred.

6.20.8 Bibliography

Hatton 18: Control flow – if structure

6.21 HFC Pointer casting and pointer type changes

[For the convenience of reviewers, here are the applicable JSF C++ rule(s):]

AV Rule 182: Type casting from any type to or from pointers shall not be used. Exception 1: Casting from void* to T* is permissible. In this case, static_cast should be used, but only if it is known that the object really is a T. Furthermore, such code should only occur in low level memory management routines. Exception 2: Conversion of literals (i.e. hardware addresses) to pointers.

AV Rule 183: Every possible measure should be taken to avoid type casting. Rationale: Errors caused by casts are among the most pernicious, particularly because they are so hard to recognize. Strict type checking is your friend – take full advantage of it.

[For the convenience of reviewers, here are the applicable MISRA 2004 rule(s)]

11.1 (req) Conversions shall not be performed between a pointer to a function and any type other than an integral type.

11.2 (req) Conversions shall not be performed between a pointer to object and any type other than an integral type, another pointer to object type or a pointer to void.

11.3 (adv) A cast should not be performed between a pointer type and an integral type.

11.4 (adv) A cast should not be performed between a pointer to object type and a different pointer to object type.

11.5 (req) A cast shall not be performed that removes any const or volatile qualification from the type addressed by a pointer.

[For the convenience of reviewers, here are the applicable CERT/CC Guideline(s)]

EXP05-A: Do not cast away a const qualification

EXP08-A: Ensure pointer arithmetic is used correctly

EXP32-C: Do not access a volatile object through a non-volatile reference

EXP34-C: Ensure a pointer is valid before dereferencing it

EXP36-C: Do not convert between pointers to objects with different alignments

[Note from Tom: after much thought and discussion, itemizing into these cases does not provide additional clarity IMHO]

6.21.0 Status and history

2008-01-25, edited by Plum
2007-11-26, reformatted by Benito
2007-11-24, edited by Moore
2007-11-24, edited by Plum
2007-10-28, edited by Plum

6.21.1 Description of application vulnerability

Define “access via a data pointer” to mean “fetch or store indirect through that pointer”; define “access via a function pointer” to mean “invocation indirect through that pointer”. The code produced for access via a pointer requires that the type of the pointer is appropriate for the data or function being accessed; otherwise undefined behavior can occur. (The detailed requirements for “appropriate” type vary among languages.)

Even if the type of the pointer is appropriate for the access, erroneous pointer operations can still cause a bug. Here is an example from CWE 188:

```
void example() {
  char a; char b; *(&a + 1) = 0;
}
```

Here, `b` may not be one byte past `a`. It may be one byte in front of `a`. Or, they may have three bytes between them because they get aligned to 32-bit boundaries.

6.21.2 Cross reference

CWE
    136: Type Errors
    188: Reliance on Data Layout

Hatton 13: Pointer casts
MISRA C 11.1, 11.2, 11.3, 11.4, add-in 11.5: Pointer casts
JSF AV 182, 183: Pointer casts
CERT/CC guidelines EXP05-A, 08-A, 32-C, 34-C and 36-C

6.21.3 Categorization

[tbd].

6.21.4 Mechanism of failure

If a pointer’s type or value is not appropriate for the data or function being accessed, erroneous behavior or undefined behavior can be the result. In particular, the last step in execution of a malicious payload is typically an invocation via a pointer-to-function which has been manipulated to point to the payload.

6.21.5 Applicable language characteristics

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

- Pointers (and/or references) can be converted to different types.
- Pointers to functions can be converted to pointers to data.
- Addresses of specific elements can be calculated.
- Integers can be added to, or subtracted from, pointers, thereby designating different objects.

6.21.6 Avoiding the vulnerability or mitigating its effects

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

[Note from Tom: no, this rule would technically be undefined behavior in C/C++, so it can't be proposed by an OWGV doc]

- Treat the compiler's pointer-conversion warnings as serious errors.
- Adopt programming guidelines (preferably augmented by static analysis) that restrict pointer conversions. For example, consider the rules itemized above from JSF AV, CERT/CC, Hatton, or MISRA C.
- Other means of assurance might include proofs of correctness, analysis with tools, verification techniques, etc.

[Note from Tom: all languages known to us already implement this rule in the language standard, so why should OWGV "reinforce" the same rule?]

[Note from Tom: not sure what this was meant to say?]

6.21.7 Implications for standardization

[tbd]

6.21.8 Bibliography

Hatton 13: Pointer casts

6.22 JCW Operator precedence/Order of Evaluation

[For the convenience of reviewers, the applicable JSF C++ rule is quoted below:

AV Rule 213: No dependence shall be placed on C++'s operator precedence rules, below arithmetic operators, in expressions.

Rationale: Readability.

From JSF C++ Appendix (with examples)

AV Rule 213

Parentheses should be used to clarify operator precedence rules to enhance readability and reduce mistakes. However, overuse of parentheses can clutter an expression thereby reducing readability. Requiring parenthesis below arithmetic operators strikes a reasonable balance between readability and clutter.

Table 2 documents C++ operator precedence rules where items higher in the table have precedence over those lower in the table.

Examples: Consider the following examples. Note that parentheses are required to specify operator ordering for those operators below the arithmetic operators.

```
x = a * b + c;           // Good: can assume "*" binds before "+"
x = v->a + v->b + w.c;  // Good: can assume "->" and "." Bind before "+"
x = (f()) + ((g()) * (h()));  // Bad: overuse of parentheses. Can assume
               //          function call binds before "+" and "*"
```

```
        x = a & b | c;           // Bad: must use parenthesis to clarify order
                        // [Note from Tom: to clarify "binding" not "order"
        x = a >> 1 + b;      // Bad: must use parenthesis to clarify order
                        // [Note from Tom: to clarify "binding" not "order"
```

[For the convenience of reviewers, I have paraphrased relevant rules from MISRA 2004]

12.1 (adv) Limited dependence should be placed on C's operator precedence rules in expressions.

[Note from Tom: MISRA rules 12.5, 12.6 and 13.2 don't really belong here or in SAM either: 12.5 (req) The operands of logical operators (&&, || and !) should be effectively Boolean; 12.6 (adv) Expressions that are effectively Boolean should not be used as operands to operators other than (&&, || and !); 13.2 (adv) Tests of a value against zero should be made explicit, unless the operand is effectively Boolean. Hard to know what to recommend. MISRA refers to an out-of-date C standard, while the current C standard (C99, and also the C++ standard C++03) incorporated an explicit Boolean type. OTOH, "use of assignment in Boolean tests" is still a vulnerability, in C and C++, even with an explicit Boolean type. I added these rules to the list in BRS "Leveraging experience and expertise" … it's one of those observations made by senior techs regarding common mistakes.]

[For the convenience of reviewers, the applicable CERT/CC Guidelines are quoted below]
EXP00-A Use parentheses for precedence of operation

6.22.0 Status and history

2008-01-21: Revised by Tom Plum [I ended up merging MTW here, and leaving SAM as a separate topic.]
2007-12-12: Reviewed at OWGV meeting 7: The existing material here probably belongs in either SAM or MTW.
2007-11-26, reformatted by Benito
2007-11-01, edited by Larry Wagoner
2007-10-15, decided at OWGV Meeting 6: We decide to write three new descriptions: operator precedence, JCW; associativity, MTW; order of evaluation, SAM. Deal with MISRA 2004 rules 12.1 and 12.2; JSF C++ rules 204, 213. Should also deal with MISRA 2004 rules 12.5, 12.6 and 13.2.

6.22.1 Description of application vulnerability

Each language provides rules of precedence and associativity, which determine, for each expression in source code, a specific syntax tree of operators and operands. These rules are also known as the rules of "grouping" or "binding"; they determine which operands are bound to each operator.

The way in which operators or sub-expressions are grouped can differ from the grouping that was expected by the programmer, causing expressions to evaluate to unexpected values. 6.22.2 Cross reference

CWE:
MISRA: 12.1
JSF: 213
CERT/CC Guidelines: EXP00-A

6.22.3 Categorization

See clause 5

6.22.4 Mechanism of failure

[Note from Tom: For C/C++, this was not correctly analyzed; if a different language was intended, I'm not able to comment … but the example appears to be C/C++.]

In C and C++, the bitwise operators (bitwise logical and bitwise shift) are sometimes thought of by the programmer as being similar to arithmetic operations, so just as one might correctly write "`x - 1 == 0`" ("`x` minus one is equal to zero"), a programmer might erroneously write "`x & 1 == 0`", mentally thinking "`x` anded-with `1` is equal to zero", whereas the operator precedence rules of C and C++ actually bind the expression as "compute `1==0`, producing 'false' i.e. zero, then bitwise-and that zero with `x`", producing (a constant) zero, contrary to the programmer's intent.

Examples from an opposite extreme can be found in programs written in APL, which is noteworthy for the absence of *any* distinctions of precedence. One commonly-made mistake is to write "a χ b + c", intending to produce "a times b plus c", whereas APL's uniform right-to-left associatively produces "c plus b, times a".

6.22.5 Applicable language characteristics

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

- Languages that permit undefined or incomplete operator precedence definitions

  [Note from Tom: are there any such languages?]
- Languages whose precedence rules are sometimes overlooked or confused by working programmers (i.e., most languages)

  [Note: moved to SAM]

6.22.6 Avoiding the vulnerability or mitigating its effects

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

- Adopt programming guidelines (preferably augmented by static analysis). For example, consider the rules itemized above from JSF C++, CERT/CC or MISRA C.

6.22.7 Implications for standardization

[Note: moved to SAM]

6.22.8 Bibliography

6.23 KOA Likely incorrect expressions

6.23.0 Status and history

2008-01-10 Minor edit by Larry Wagoner
2007-12-15: Minor editorial cleanup by Moore
2007-11-26, reformatted by Benito
2007-10-29, edited by Larry Wagoner
2007-10-15, OWGV Meeting 6 decided that: "We should introduce a new item, KOA, for code that executes with no result because it is a symptom of misunderstanding during development or maintenance. (Note that this is similar to unused variables.) We probably want to exclude cases that are obvious, such as a null statement, because they are obviously intended. It might be appropriate to require justification of why this has been done. These may turn out to be very specific to each language. The rule needs to be generalized. Perhaps it should be phrased as statements that execute with no effect on all possible execution paths. It should deal with MISRA rules 13.1, 14.2, 12.3 and 12.4. Also MISRA rule 12.13. It is related to XYQ but different. "

6.23.1 Description of application vulnerability

Certain expressions are symptomatic of what is likely a mistake by the programmer. The statement is not wrong, but it is unlikely to be right. The statement may have no effect and effectively is a null statement or may introduce an unintended vulnerability. A common example is the use of "=" in an `if` expression in C where the programmer meant to do an equality test using the "==" operator. Other easily confused operators in C are the logical operators such as && for the bitwise operator &. It is legal and possible that the programmer intended to do an assignment within the if expression, but due to this being a common error, a programmer doing so would be using a poor programming practice. A less likely occurrence, but still possible is the substitution of "==" for "=" in what is supposed to be an assignment statement, but which effectively becomes a null statement. These mistakes may survive testing only to manifest themselves or even be exploited as a vulnerability under certain conditions.

6.23.2 Cross reference

CWE: 480, 481, 482, 570, 571
JSF: 160, 166
MISRA: 12.3, 12.4, 12.13, 13.1, 14.2

6.23.3 Categorization

See clause 5

6.23.4 Mechanism of failure

Some of the failures are simply a case of programmer carelessness. Substitution of "=" instead of "==" in a Boolean test is easy to do and most C/C++ programmers have made this mistake at one time or another. Other instances can be the result of intricacies of compilers that affect whether statements are optimized out. For instance, having an assignment expression in a Boolean statement is likely making an assumption that the complete expression will be executed in all cases. However, this is not always the case as sometimes the truth value of the Boolean expression can be determined after only executing some portion of the expression. For instance:

```
        if ((a == b) || (c = (d-1)))
```

There is no guarantee which of the two subexpressions `(a == b) or (c=(d-1))` will be executed first. Should `(a==b)` be determined to be true, then there is no need for the subexpression `(c=(d-1))` to be executed and as such, the assignment `(c=(d-1))` will not occur.

Embedding expressions in other expressions can yield unexpected results. Putting an expression as the argument for a function call will likely not execute the expression, but simply use it as the value to be passed to the function. Increment and decrement operators (++ and – –) can also yield unexpected results when mixed into a complex expression.

6.23.5 Applicable language characteristics

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

- All languages are susceptible to likely incorrect expressions.

6.23.6 Avoiding the vulnerability or mitigating its effects

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

- Simplify expressions. Attempting to perform very sophisticated expressions that contain many subexpressions can look very impressive. It can also be a nightmare to maintain and to understand for subsequent programmers who have to maintain or modify it. Striving for clarity and simplicity may not look as impressive, but it will likely make the code more robust and definitely easier to understand and debug.
- Do not use assignment expressions as function parameters. Sometimes the assignment may not be executed as expected. Instead, perform the assignment before the function call.
- Do not perform assignments within a Boolean expression. This is likely unintended, but if not, then move the assignment outside of the Boolean expression for clarity and robustness.

- On some rare occasions, some statements intentionally do not have side effects and do not cause control flow to change. These should be annotated through comments and made obvious that they are intentionally no-ops with a stated reason. If possible, such reliance on null statements should be avoided. In general, except for those rare instances, all statements should either have a side effect or cause control flow to change.

6.23.7 Implications for standardization

None.

6.23.8 Bibliography

6.24 MEM Deprecated Language Features

6.24.0 Status and history

2008-01-10 Minor edit by Larry Wagoner
2007-12-15 Minor editorial cleanup by Jim Moore
2007-11-26, reformatted by Benito
2007-11-01, edited by Larry Wagoner
2007-10-15, created by OWG Meeting #6. The following content is planned:
Create a new description for deprecated features, MEM. This might be focal point of a discussion of what to do when your language standard changes out from underneath you. Include legacy features for which better replacements exist. Also, features of languages (like multiple declarations on one line) that commonly lead to errors or difficulties in reviewing. The generalization is that experts have determined that use of the feature leads to mistakes.
Include MISRA 2004 rules 1.1, 4.2; JSF C++ rules 8, 152.

6.24.1 Description of application vulnerability

All code should conform to the current standard for the respective language. In reality though, a language standard may change during the creation of a software system or suitable compilers and development environments may not be available for the new standard for some period of time after the standard is published. In order to smooth the process of evolution, features that are no longer needed or which serve as the root cause of or contributing factor for safety or security problems are often deprecated to temporarily allow their use but to indicate that those features will be removed in the future. The deprecation of a feature is a strong indication that it should not be used. Other features, although not formally deprecated, are rarely used and there exists other alternative and more common ways of expressing the same function. Use of these rarely used features can lead to problems when others are assigned the task of debugging or modifying the code containing those features.

6.24.2 Cross reference

CWE:
JSF: 8, 152
MISRA 2004: 1.1, 4.2

6.24.3 Categorization

See clause 5.?.

6.24.4 Mechanism of failure

Most languages evolve over time. Sometimes new features are added making other features extraneous. Languages may have features that are frequently the basis for security or safety problems. The deprecation of these features indicates that there is a better way of accomplishing the desired functionality. However, there is always a time lag between the acknowledgement that a particular feature is the source of safety or security problems, the decision to remove or replace the feature and the generation of warnings or error messages by compilers that the feature shouldn't be used. Given that software systems can take many years to develop, it is possible and even likely that a language standard will change causing some of the features used to be suddenly deprecated. Modifying the software can be costly and time consuming to remove the deprecated features. However, if the schedule and resources permit, this would be prudent as future vulnerabilities may result from leaving the deprecated features in the code. Ultimately the deprecated features will likely need to be removed when the features are removed. Removing the features sooner rather than later would be the best course of action to take.

6.24.5 Applicable language characteristics

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

- All languages
    - that have standards, though some only have defacto standards.
    - that evolve over time and as such could potentially have deprecated features at some point.

6.24.6 Avoiding the vulnerability or mitigating its effects

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

- Rarely used or complicated features of a language should not be used as peer review and future maintenance could inadvertently introduce vulnerabilities due to a lack of complete understanding of obscure features of a language. The skill level of those who eventually modify or maintain the code or reuse the code cannot be guaranteed. Keeping constructs simple can make future code debugging, reuse and enhancements easier and more successful.
- Adhere to the latest published standard for which a suitable complier and development environment is available
- Avoid the use of deprecated features of a language
- Avoid the use of complicated features of a language
- Avoid the use of rarely used constructs that could be difficult for entry level maintanence personnel to understand
- Stay abreast of language discussions in language user groups and standards groups on the Internet. Discussions and meeting notes will give an indication of problem prone features that should not be used or used with caution.

6.24.7 Implications for standardization

- Obscure language features for which there are commonly used alternatives should be considered for removal from the language standard.
- Complicated features which have been routinely been found to be the root cause of safety or security vulnerabilities or which are routinely disallowed in software guidance documents should be considered for removal from the language standard.

6.24.8 Bibliography

6.25 NMP Pre-processor Directives

6.25.0 Status and history

2007-11-19, Edited by Benito
2007-10-15, Decided at OWGV meeting #6: "Write a new description, NMP about the use of preprocessors directives and the increased cost of static analysis and the readability difficulties. MISRA C:2004 rules in 19 and JSF rules from 4.6 and 4.7.

6.25.1 Description of application vulnerability

Pre-processor replacements happen before any source code syntax check, therefore there is no type checking – this is especially important in function-like macro parameters.

If great care is not taken in the writing of macros, the expanded macro can have an unexpected meaning. In many cases if explicit delimiters are not added around the macro text and around all macro arguments within the macro text unexpected expansion is the results.

Source code that relies heavily on complicated pre-processor directives may result in obscure and hard to maintain code since the syntax they expect is on many occasions different from the regular expressions programmers expect in the programming language that the code is written.

6.25.2 Cross reference

CWE: none
Holtzmann-8
JSF: 26, 27, 28, 29, 30, 31, and 32
MISRA: 19.7, 19.8, and 19.9

6.25.3 Categorization

See clause 5.?.

6.25.4 Mechanism of failure

Readability and maintainability is greatly increased if the language features available in the programming language are used instead of a pre-processor directive.

Static analysis while can identify many problems early; heavy use of the pre-processor can limit the effectiveness of many static analysis tools.

In many cases where complicated macros are used, the program does not do what is intended. For example:

define a macro as follows,

```
#define CD(x, y) (x + y - 1) / y
```

whose purpose is to divide. Then suppose it is used as follows

```
a = CD (b & c, sizeof (int));
```

this will normally expands into

```
a = (b & c + sizeof (int) - 1) / sizeof (int);
```

which most times will not do what is intended. Defining the macro as

```
#define CD(x, y) ((x) + (y) - 1) / (y)
```

will normally provide the desired result.

6.25.5 Applicable language characteristics

- Unintended groupings of arithmetic statements
- Improperly nested language constructs
- Cascading macros
- Duplication of side effects

- Macros that reference themselves
- Nested macro calls
- Reliance on complicated macros

6.25.6 Avoiding the vulnerability or mitigating its effects

All functionality that can be accomplished without the use of a pre-processor should be used before using a pre-processor.

6.25.7 Implications for standardization

6.25.8 Bibliography

6.26 NYY Dynamically-linked code and self-modifying code (was Self-modifying Code)

6.26.0 Status and history

2008-01-22, edited by Plum
2007-12-13, considered at OWGV meeting 7
2007-22-16, reformatted by Benito
2007-11-22, edited by Plum

6.26.1 Description of application vulnerability

On some platforms, and in some languages, instructions can modify other instructions in the code space ("self-modifying code").

[Note from Tom: could we just drop this DLL topic? It seems rather tangential to language vulnerabilities …]

6.26.2 Cross reference

JSF AV rule 2: No self-modifying code.

6.26.3 Categorization

[tbd].

6.26.4 Mechanism of failure

On some platforms, a pointer-to-data can (erroneously) be given an address value that designates a location in the instruction space. If subsequently a modification is made through that pointer, then a critical undefined behavior can result.

6.26.5 Applicable language characteristics

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

- Pointer-to-data can be given an address value that designates a location in the instruction space
- Any other method to create self-modifying code;

6.26.6 Avoiding the vulnerability or mitigating its effects

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

- Avoid implementation languages that allow self-modifying code.

- Each assignment to, or modification of, a pointer should be verified to be semantically correct.

6.26.7 Implications for standardization

[tbd]

6.26.8 Bibliography

6.27 PLF Floating Point Arithmetic

6.27.0 Status and history

2008-01-10 Edited by Larry Wagoner
2007-12-15: Minor editorial cleanup, Jim Moore
2007-11-26, reformatted by Benito
2007-10-30, edited by Larry Wagoner
2007-10-15, decided at OWGV Meeting #6: " Add to a new description PLF that says that when you use floating point, get help. The existing rules should be cross-referenced. MISRA 2004 rules 13.3, 13.4, add-in 1.5, 12.12; JSF rule 184."

6.27.1 Description of application vulnerability

Only a relatively small proportion of real numbers can be represented exactly in a computer. To represent real numbers, most computers use ANSI/IEEE Std 754. The bit representation for a floating point number can vary from compiler to compiler and on different platforms. Relying on a particular representation can cause problems when a different compiler is used or the code is reused on another platform. Regardless of the representation, many real numbers can only be approximated since representing the real number using a binary representation would require an endlessly repeating string of bits or more binary digits than are available for representation. Therefore it should be assumed that a floating point number is only an approximation, even though it may be an extremely good one. Floating point representation of a real number or a conversion to floating point can cause surprising results and unexpected consequences to those unaccustomed to the idiosyncrasies of floating point arithmetic.

6.27.2 Cross reference

CWE: none
JSF: 146, 147, 184, 197, 202
MISRA: 13.3, 13.4

6.27.3 Categorization

See clause 5.?.

6.27.4 Mechanism of failure

Floating point numbers are generally only an approximation of the actual value. In the base 10 world, the value of 1/3 is 0.333333… The same type of situation occurs in the binary world, but numbers that can be represented with a limited number of digits in base 10, such as 1/10=0.1 become endlessly repeating sequences in the binary world. So 1/10 represented as a binary number is:

0.0001100110011001100110011001100110011001100110011…

Which is 0*1/2 + 0*1/4 + 0*1/8 + 1*1/16 + 1*1/32 + 0*1/64… and no matter how many digits are used, the representation will still only be an approximation of 1/10. Therefore when adding 1/10 ten times, the final result may or may not be exactly 1.

Using a floating-point variable as a loop counter can propagate rounding and truncation errors over many iterations so that unexpected results can occur. Rounding and truncation can cause tests of floating point numbers against other values to yield unexpected results. One of the largest manifestations of floating point errors is reliance upon comparisons of floating point values. Tests of equality/inequality can vary due to propagation or conversion errors. Differences in magnitudes of floating point numbers can result in no change of a very large floating-point number when a relatively small number is added to or subtracted from it. These and other idiosyncrasies of floating point arithmetic require that users of floating-point arithmetic be very cautious in their use of it.

Manipulating bits in floating point numbers is also very implementation dependent. Though IEEE 754 is a commonly used representation for floating point data types, it is not universally used or required by all computer languages. Some languages predate IEEE 754 and make the standard optional. IEEE 754 uses a 24 bit mantissa (including the sign bit) and an 8 bit exponent, but the number of bits allocated to the mantissa and exponent can vary when using other representations as can the particular representation used for the mantissa and exponent. Typically special representations are specified for positive and negative zero and infinity. Relying on a particular bit representation in inherently problematic, especially when a new compiler is introduced or the code is reused on another platform.

6.27.5 Applicable language characteristics

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

- All languages with floating point variables can be subject to rounding or truncation errors

6.27.6 Avoiding the vulnerability or mitigating its effects

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

- Do not use a floating point expression in a Boolean test for equality. Instead of an expression, use a library that determines the difference between the two values to determine whether the difference is acceptably small enough so that two values can be considered equal. Note that if the two values are very large, the “small enough” difference can be a very large number.
- Avoid the use of a floating point variable as a loop counter. If necessary to use a floating point value as a loop control, use inequality to determine the loop control (i.e. <, <=, > or >=)
- Understand the floating-point format used to represent the floating-point numbers. This will provide some understanding of the underlying idiosyncrasies of floating point arithmetic.
- Manipulating the bit representation of a floating point number should not be done except with built-in operators and functions that are designed to extract the mantissa and exponent.

6.27.7 Implications for standardization

- Do not use floating-point for exact values such as monetary amounts. Use floating point only when necessary such as for fundamentally inexact values such as measurements.
- Languages that do not already adhere to or only adhere to a subset of ANSI/IEEE 754 should consider adhering completely to the standard. Note that ANSI/IEEE 754 is currently undergoing revision as ANSI/IEEE 754r and comments regarding 754 refer to either 754 or the new 754r standard when it is approved. Examples of standardization that should be considered:
  - C, which predates ANSI/IEEE Std 754 and currently has it as optional in C99, should consider requiring ANSI/IEEE 754 for floating point arithmetic
  - Java should consider fully adhering to ANSI/IEEE Std 754 instead of only a subset
- All languages should consider standardizing their data types on ISO/IEC 10967-3:2006

6.27.8 Bibliography

[1] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[2] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-1985. Institute of Electrical and Electronics Engineers, New York, 1985.

[3] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 http://www.nsc.liu.se/wg25/book

[4] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, http://archive.gao.gov/t2pbat6/145960.pdf

[5] Robert Skeel, *Roundoff Error Cripples Patriot Missile*, SIAM News, Volume 25, Number 4, July 1992, page 11, http://www.siam.org/siamnews/general/patriot.htm

[6] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf (Press release is at: http://www.esa.int/esaCP/Pr_33_1996_p_EN.html and there is a link to the report at the bottom of the press release)

[7] ISO/IEC 10967-3:2006. ISO/IEC Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions, ISO/IEC Standard 10967-3:2006, International Organization for Standardization/International Electrotechnical Commission, May 2006 http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=37994

6.28 RVG Pointer Arithmetic

6.28.0 Status and history

2007-11-19 Edited by Benito
2007-10-15, Decided at OWGV meeting #6: "Write a new description RVG for Pointer Arithmetic, for MISRA C:2004 17.1 thru 17.4."

6.28.1 Description of application vulnerability

Using pointer arithmetic incorrectly can lead to miscalculations that can result in serious errors, buffer overflows and underflows, and addressing arbitrary memory locations.

6.28.2 Cross reference

CWE: none
JSF: 215
MISRA: 17.1, 17.2, 17.3, and 17.4

6.28.3 Categorization

See clause 5.?.

6.28.4 Mechanism of failure

Pointer arithmetic used incorrectly can produce:

- Buffer overflow
- Buffer underflow
- Addressing arbitrary memory locations
- Addressing memory outside the range of the program

6.28.5 Applicable language characteristics

This vulnerability description is intended to be applicable to languages that allow pointer arithmetic.

6.28.6 Avoiding the vulnerability or mitigating its effects

- Use pointer arithmetic only for indexing objects defined as arrays.
- Use only an integer for addition and subtraction of pointers

6.28.7 Implications for standardization

6.28.8 Bibliography

6.29 STR Bit Representations

6.29.0 Status and history

2008-0110 Edited by Larry Wagoner
2007-12-15: minor editorial cleanup, Jim Moore
2007-11-26, reformatted by Benito
2007-11-01, edited by Larry Wagoner
2007-10-15, decided at OWGV Meeting #6: Write a new vulnerability description, STR, that deals with bit representations. It would say that representations of values are often not what the programmer believes they are. There are issues of packing, sign propagation, endianness and others. Boolean values are a particular problem because of packing issues. Programmers who depend on the bit representations of values should either utilize language facilities to control the representation or document that the code is not portable. MISRA 2004 rules 6.4, 6.5, add-in 3.5, 12.7.

6.29.1 Description of application vulnerability

Computer languages frequently provide a variety of sizes for integer variables. Languages may support short, integer, long, and even big integers. Interfacing with protocols, device drivers, embedded systems, low level graphics or other external constructs may require each bit or set of bits to have a particular meaning. Those bit sets may or may not coincide with the sizes supported by a particular language. When they do not, it is common practice to pack all of the bits into one word. Masking and shifting of the word using powers of two to pick out individual bits or using sums of powers of 2 to pick out subsets of bits (e.g. using 28=2^2+2^3+2^4 to create the mask 11100 and then shifting 2 bits) provides a way of extracting those bits. Knowledge of the underlying bit storage is usually not necessary to accomplish simple extractions such as these. Problems can arise when programmers mix their techniques to reference the bits or output the bits. The storage ordering of the bits may not be what the programmer expects when writing out the integers which contain the words.

6.29.2 Cross reference

CWE:
JSF: 147, 155
MISRA: 3.5, 6.4, 6.5, 12.7

6.29.3 Categorization

See clause 5.?. <*Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*>

6.29.4 Mechanism of failure

Packing of bits in an integer is not inherently problematic. However, an understanding of the intricacies of bit level programming must be known. One problem arises when assumptions are made when interfacing with outside constructs and the ordering of the bits or words are not the same as the receiving entity. Programmers may inadvertently use the sign bit in a bit field and then may not be aware that an arithmetic shift (sign extension) is being performed when right shifting causing the sign bit to be extended into other fields. Alternatively, a left shift can cause the sign bit to be one. Some computers or other devices store the bits left to right while others store them right to left. The type of storage can cause problems when interfacing with outside devices that expect the bits in the opposite order. Bit manipulations can also be problematic when the manipulations are done on binary encoded records that span multiple words. The storage and ordering of the bits must be considered when doing bitwise operations across multiple words as bytes may be stored in big endian or little endian format.

6.29.5 Applicable language characteristics

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

- Languages that allow bit manipulations
- Languages that are commonly used for protocol encoding/decoding, device drivers, embedded system programming, low level graphics or other low level programming
- Language that permit bit fields

6.29.6 Avoiding the vulnerability or mitigating its effects

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

- Bit meanings should be explicitly documented along with any assumptions about bit ordering
- The way bit ordering is done on the host system and on the systems with which the bit manipulations will be interfaced should be understood
- Bit fields should be used in languages that support them
- Bit operators should not be used on signed operands

6.29.7 Implications for standardization

- For languages that are commonly used for bit manipulations, an API for bit manipulations that is independent of word length and machine instruction set should be defined and standardized.

6.29.8 Bibliography

[1] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 http://www.embedded.com/1999/9907/9907feat2.htm

6.30 TRJ Use of Libraries

6.30.0 Status and history

2007-11-19, Edited by Benito
2007-10-15, Decided at OWGV meeting #6: "Write a new item, TRJ. Calls to system functions, libraries and APIs might not be error checked. It may be necessary to perform validity checking of parameters before making the call."

6.30.1 Description of application vulnerability

Libraries that supply objects or functions are in most cases not required to check the parameters passed to the function or object to be valid. In those cases where parameter validation is required there might not be adequate parameter validation.

6.30.2 Cross reference

CWE: 114
JSF: 16, 18, 19, 20, 21, 22, 23, 24, and 25
MISRA: 20.2, 20.3, 20.4, 20.6, 20.7, 20.8, 20.9, 20.10, 20.11, and 20.12

6.30.3 Categorization

See clause 5.?.

6.30.4 Mechanism of failure

Undefined behaviour.

6.30.5 Applicable language characteristics

This vulnerability description is intended to be applicable to Libraries that do not validate the parameters accepted by functions, methods and objects.

6.30.6 Avoiding the vulnerability or mitigating its effects

There are several approaches that can be taken, some work best if used in conjunction with each other.

- Validate the values passed before the value is used.
- Use only libraries that have been validated to perform the needed checks. For example use only libraries that are DO-178B level A certified.
- Develop wrappers around library functions that check the parameters before calling the function.
- Demonstrate statically that the parameters are never invalid.
- Use only libraries written in-house and have been developed with safety-critical requirements.

6.30.7 Implications for standardization

- All languages that define a support library should consider removing most if not all cases of undefined behaviour from the library sections.
- Define the libraries so that all parameters are validated.

6.30.8 Bibliography

Holtzmann-7

6.31 FAB Implementation-defined behavior

6.31.0 Status and history

2008-02-11, Revised by Derek Jones
2007-12-12: Considered at OWGV meeting 7: See notes added to BQF. Consider issues arising from maintenance that might involve changes in the selected implementation.
2007-10-15, Jim Moore added notes from OWGV Meeting #6: "So-called portability issues should be confined to EWF, BQF, and FAB. The descriptions should deal with MISRA 2004 rules 1.2, 3.1, 3.2, 3.3, 3.4 and 4.a; and JSF C++ rules 210, 211, 212, 214. Also discuss the role of pragmas and assertions."
2007-07-18, Edited by Jim Moore
2007-06-30, Created by Derek M. Jones

6.31.1 Description of application vulnerability

The external behavior of a program, whose source code contains one or more instances of constructs having implementation-defined behavior, when the source code is recompiled or relinked.

6.31.2 Cross reference

Ada: Clause 1.1.3 Conformity of an Implementation with the Standard; Clause 3.4.1 implementation-defined behavior
C: Clause 3.4.1 implementation-defined behavior
C++: Clause 1.3.5 implementation-defined behavior
Fortran: Clause 1.5 Conformance (Fortran uses the term 'processor dependent')
Also see guideline recommendations: BQF-071212-unspecified-behavior and EWF-undefined-behavior.

6.31.3 Categorization

See clause 5.1.2.

6.31.4 Mechanism of failure

Language specifications do not always uniquely define the behavior of a construct. When an instance of a construct that is not uniquely defined is encountered (this might be at any of compile, link, or run time) implementations are permitted to choose from a set of behaviors. The only difference from unspecified behavior is that implementations are required to document how their they behave.

A developer may use a construct in a way that depends on a particular implementation-defined behavior occurring. The behavior of a program containing such a usage is dependent on the translator used to build it always selecting the 'expected' behavior.

Some implementations provide a mechanism for changing an implementation's implementation-defined behavior (e.g., use of pragmas in source code). Use of such a change mechanism creates the potential for additional human error in that a developer may be unaware that a change of behavior was requested earlier in the source code and may write code that depends on the previous, unchanged, implementation-defined behavior.

6.31.5 Interrupting the failure mechanism

Many language constructs may have implementation-defined behavior and unconditionally recommending against any use of these constructs may be completely impractical. For instance, in many languages the number of significant characters in an identifier is implementation-defined (and it is not possible to write useful programs without using identifiers)

In the identifier significant character example developers must choose a minimum number of characters and require that only translators supporting at least that number, *N*, of characters be used.

The appearance of implementation-defined behavior in a language specification is a recognition by the designers that in some cases implementation flexibility provides a worthwhile benefit for language translators; this usage is not a defect in the language.

6.31.6 Assumed variations among languages

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

- languages whose specification allows some variation in how a translator handles some construct, where reliance on one form of this variation can result in differences in external program behavior.
- Implementations may not be required to provide a mechanism for controlling implementation-defined behavior.

6.31.7 Avoiding the vulnerability or mitigating its effects

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

- Ensuring that a specific use of a construct having implementation-defined behavior produces an external behavior that is the same, for that specific use, for all of possible behaviors permitted by the language specification.
- Only use a language implementation whose implementation-defined behaviors are within a known subset of implementation-defined behaviors. The known subset being chosen so that the 'same external behavior' condition described above is met.
- Create very visible documentation (e.g., at the start of a source file) that the default implementation-defined behavior is changed within the current file[Other recommendations ???]

Portability guidelines for a specific language may provide a list of common implementation behaviors.

When developing coding guidelines for a specific language all constructs that have implementation-defined behavior shall be documented and for each construct the situations where the set of possible behaviors can varied shall be enumerated.

When applying this guideline on a project the functionality provided by an for changing its implementation-defined behavior shall be documented [and ???].

6.31.8 Bibliography

6.32 NAI Choice of Clear Names

6.32.0 History

2008-01-10 Minor edit by Larry Wagoner
2007-12-13, Considered at OWGV 7: Minor changes suggested
2007-11-26 Edited by Larry Wagoner
2007-10-15 May need more work by Steve Michell to incorporate this decision of OWGV meeting 6: Write a new description, NAI, on issues in selecting names. Assign this one to Steve Michell. Look at Derek's paper on the subject. Deal with JSF rules 48-56.
2007-10-03 Edited by OWGV Meeting #6
2007-10-02 Contributed by Steve Michell

6.32.1 Description

Humans sometimes choose similar or identical names for objects, types, aggregates of types, subprograms and modules. They tend to use characteristics that are specific to the native language of the software developer to aid in this effort, such as use of mixed-casing, underscores and periods, or use of plural and singular forms to support the separation of items with similar names. Similarly, development conventions sometimes use casing (e.g. all uppercase for constants, etc).

Human cognitive problems occur when different (but similar) objects, subprograms, types, or constants differ in name so little that human reviewers are unlikely to distinguish between them, or when the system maps such entities to a single entity.

Conventions such as the use of text case, and singular/plural distinctions may work in small and medium projects, but there are a number of significant issues to be considered:

- Large projects often have mixed languages and such conventions are often language-specific
- Today's identifiers can be international and some language character sets have different notions of casing and plurality
- Different word-forms tend to be language-specific (e.g. English) and may be meaningless to humans from other dialects

An important general issue is the choice of names that differ from each other negligibly (in human terms), for example by differing by only underscores, (none, "_" "__", ...), plurals ("s"), visually identical letters (such as "l" and "1", "O" and "0"), or underscores/dashes ("-","_"). [There is also an issue where identifiers appear distinct to a human but identical to the computer, e.g. FOO, Foo, and foo in some languages. Character sets extended with diacritical marks and non-Latin characters may offer additional problems. Some languages or their implementations may pay attention to only the first n characters of an identifier.

There are a couple of similar situations that may occur, but which are notably different. This is different than overloading or overriding where the same name is used intentionally (and documented) to access closely linked sets of subprograms.   This is also different than using reserved names which can lead to a conflict with the reserved use and the use of which may or may not be detected at compile time.

Although most such mistakes are unintentional, it is plausible that such mistakes can be intentional, if masking surreptitious behaviour is a goal.

6.32.2 Cross Reference

JSF-C++ : Rules 48-56

6.32.3 Categorization

6.32.4 Mechanism of Failure

- Calls to the wrong subprogram or references to the wrong data element (that was missed by human review) can cause unintended behaviour. Language processors will not make a mistake in name translation, but human cognition limitations may cause humans to misunderstand, and therefore may be easily missed in human reviews.

6.32.5 Applicable language characteristics

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

- Languages with relatively flat name spaces will be more susceptible. Systems with modules, classes, packages can qualify names to disambiguate names that originate from different parents.
- Languages that provide preconditions, postconditions, invariances and assertions or redundant coding of subprogram signatures help to ensure that the subprograms in the module will behave as expected, but do nothing if different subprograms are called.
- Languages that treat letter case as significant. Some languages do not differentiate between names with differing case, while others do.

6.32.6 Avoiding the Vulnerability or Mitigating its Effects

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

- Implementers can create coding standards that provide meaningful guidance on name selection and use. Good language specific guidelines could eliminate most problems.
- Use static analysis tools to show the target of calls and accesses and to produce alphabetical lists of names. Human review can then often spot the names that are sorted at an unexpected location or which look almost identical to an adjacent name in the list.
- Use static tools (often the compiler) to detect declarations that are unused.
- Use languages with a requirement to declare names before use or use available tool or compiler options to enforce such a requirement.

6.32.7 Implications for Standardization

- Languages that do not require declarations of names should consider providing an option that does impose that requirement.

6.32.8 References

Jones, Derek, "Some proposed language vulnerability guidelines" Submitted to the December 2006 Washington, D.C. meeting of the ISO/IEC SC22 OWGV

Jones, Derek M., "The New C Standard (Identifiers)" `www.coding-guidelines.com/cbook/sent792.pd`

JSF C++

6.33 AJN Choice of Filenames and other External Identifiers

6.33.0 History

2008-01-10: Edited by Larry Wagoner
2007-12-13: New topic: Larry Wagoner

6.33.1 Description

Interfacing with the directory structure or other external identifiers on a system on which software executes is very common. Differences among operating systems can make this interface problematic. The directory structure, permissible characters, case sensitivity, and so forth can vary among operating systems and even among variations of the same operating system. For instance, some characters and filenames on one platform may be verbatim on another. For example, on OS X, ":" is prohibited as part of a filename. Microsoft XP prohibits "/?:&*"<>|#%". Many flavours of Unix allow any character except for the reserved character / used to delineate the directory structure.

Some operating systems are case sensitive while others are not. On non-case sensitive operating systems, depending on the software being used, the same filename could be displayed as filename, Filename or FILENAME and all would refer to the same file.

Some operating systems, particularly older ones, only rely on the significance of the first n characters of the file name. N can be unexpectedly small, such as the first 8 characters in the case of Win16 architectures which would cause "filename1", "filename2" and "filename3" to all map to the same file.

Being unclear as to what filename, named resource or external identifier is being referenced can be the basis for problems. Such mistakes or ambiguity can be unintentional, can be intentional or can be potentially exploited, if surreptitious behaviour is a goal.

6.33.2 Cross Reference

6.33.3 Categorization

6.33.4 Mechanism of Failure

The wrong file or named resource may be used unintentionally. Attackers could exploit this situation to intentionally misdirect access of a file or other named resource to another file or named resource.

6.33.5 Applicable language characteristics

A particular language interface to a system should be consistent in its processing of filenames or external identifiers. Consistency is only the first consideration. Even though it is consistent, it may consistently do something that is unexpected by the developer of the software interfacing with the system.

6.33.6 Avoiding the Vulnerability or Mitigating its Effects

- Use operating systems that are compliant with ISO/IEC 9945:2003 (IEEE Std 1003.1-2001). Most popular operating systems are either fully compatible or compliant via a compatibility feature. Full compliance should be preferred.
- For operating systems with a compatibility feature for ISO/IEC 9945:2003, the compatibility features should be used.

6.33.7 Implications for Standardization

- Language APIs for interfacing with external identifiers should be compliant with ISO/IEC 9945:2003 (IEEE Std 1003.1-2001).

6.33.8 References

Jones, Derek, “Some proposed language vulnerability guidelines” Submitted to the December 2006 Washington, D.C. meeting of the ISO/IEC SC22 OWGV

6.34 YOW Identifier name reuse

6.34.0 Status and history

2008-02-14, Edited by Chad Dougherty
2008-01-04 Edited by Robert C. Seacord
Pending (rewrite needed)REWRITE: Robert Seacord (references immediately below relate to N0102)
2007-10-15 Also decided at OWGV Meeting 6: "add something about issues in redefining and overloading operators – MISRA 2004 rules 5.2, 8.9, 8.10; JSF C++ rule 159".
2007-10-15 Also decided at OWGV Meeting 6: Deal with MISRA 2004 rules 5.3, 5.4, 5.5, 5.6, 5.7, 20.1, 20.2
2007-10-15 Also decided at OWGV Meeting 6: Deal with JSF C++ rule 120.
2007-10-01, Edited at OWGV Meeting #6
2007-07-19, Edited by Jim Moore
2007-06-30, Created by Derek Jones

6.34.1 Description of application vulnerability

When distinct identifiers entitiesentities are defined in nested scopes using the same name it is possible that program logic will operate on an entity other than the intended entity. For example, whenif one of the definitionsthe innermost definition is deleted from the source, the program will continue to compile without a diagnostic being issued [but execution will provide different results].

6.34.2 Cross reference

CWE: Nothing applicable
CERT C: DCL32-C,
MISRA C 2004: 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 20.1, 20.2
JSF C++: 120, 159

6.34.3 Categorization

See clause 5.2.

6.34.4 Mechanism of failure

Many languages support the concept of scope. One of the ideas behind the concept of scope is to provide a mechanism for the independent definition of identifiers that may share the same name.

For instance, in the following code fragment:

```
int some_var;

   {
   int t_var;
   int some_var; /* definition in nested scope */

   t_var=3;
   some_var=2;
   }
```

an identifier called `some_var` has been defined in different scopes.

If the either the definition of `some_var` or `t_var` that occurs in the nested scope is deleted (e.g., when the source is modified) it is necessary to delete all other references to that identifier within the scope. If a developer deletes the definition of `t_var` but fails to delete the statement that references it, then most languages require a diagnostic to be issued (e.g., reference to undefined variable). However, if the nested definition of `some_var` is deleted but the reference to it in the nested scope is not deleted, then no diagnostic will be issued (because the reference resolves to the definition in the outer scope).

Non-unique identifiers in the same scope can also be introduced through the use of identifiers whose common substring exceeds the length of characters the implementation considers to be distinct. For example, in the following code fragment:

```
    extern int global_symbol_definition_lookup_table_a[100];

    extern int global_symbol_definition_lookup_table_b[100];
```

the external identifiers are not unique on implementations where only the first 31 characters are significant.

A related problem exists in languages that allow overloading or overriding of keywords or standard library function identifiers. Such overloading can lead to confusion about which entity is intended to be referenced.

6.34.5 Interrupting the failure mechanism

New identifiers should not be defined using a name that is already visible within which the scope of the new definition. Alternately, utilize language-specific facilities that check for and prevent inadvertent overloading of names should be used.

6.34.6 Assumed variations among languages

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

- Languages which require a diagnostic to be issued if an identifier is referenced and no definition is visible for that identifier.

6.34.7 Avoiding the vulnerability or mitigating its effects

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

- Ensuring that a definition of an identifier entity does not occur in a scope where a different identifier entity with the same name is accessible. A language-specific project coding convention can be used to ensure that such errors are detectable.
- Ensuring that a definition of an identifier entity does not occur in a scope where a different identifier entity with the same name is accessible and has a type which permits it to occur in at least one context where the first identifier entity can occur.
- UtilizeUse language features, if any, that explicitly mark definitions of entities that are intended to hide other definitions.
- Overloaded operations or methods should form families that use the same semantics, share the same name, have the same purpose, and that are differentiated by formal parameters.
- Determining the number of significant characters recognized by the most restrictive implementation used and documenting this assumption in the code.

6.34.8 References

Hatton 2003

MISRA C 2004

CERT C

6.35 IHN Type system (name changed from Strong typing)

[For the convenience of reviewers, the applicable JSF C++ rules are quoted below:
[AV Rule 148 Enumeration types shall be used instead of integer types (and constants) to select from a limited series of choices.
[Note: This rule is not intended to exclude character constants (e.g. 'A', 'B', 'C', etc.) from use as case labels.
[Rationale: Enhances debugging, readability and maintenance. Note that a compiler flag (if available) should be set to generate a warning if all enumerators are not present in a switch statement.
[AV Rule 183 Every possible measure should be taken to avoid type casting.
[Rationale: Errors caused by casts are among the most pernicious, particularly because they are so hard to recognize. Strict type checking is your friend – take full advantage of it.]

[For the convenience of reviewers, I have paraphrased the applicable rules from MISRA 2004:
6.1 Use the plain type char only for character values.
6.2 Use the signed and unsigned type char only for numeric values.
6.3 In place of basic types, use typedefs that indicate size and signedness. The POSIX typedefs are recommended.

6.35.0 Status and history

REVISE: Jim Moore
2007-12-12: Considered at OWGV meeting 7. Thoughts included: Don't write the description in terms of strong/weak typing. Realistically, different languages provide different typing capabilities. // Use whatever typing facilities are available. // Code as if data is typed even if the language doesn't provide for it. // Exclude automatically generated code. // Pay attention to whatever messages the compiler generates regarding type violations. // Tom Plum offered to send more suggestions. // Erhard offered to send some examples.
2007-12-07: Formatting changes and minor improvements made by Jim Moore.
2007-10-15: OWGV Meeting 6 decided: Write a new description, IHN, to encourage strong typing but deal with performance implications. Use enumeration types when you intend to select from a manageably small set of alternatives. Deal with issues like char being implementation-defined in C. Discuss how one should introduce names (e.g. typedefs) to document typing decisions and check them with tools. Deal with MISRA 2004 rules 6.1, 6.2, 6.3; JSF rules 148, 183.

6.35.1 Description of application vulnerability

When data values are converted from one type to another, even when done intentionally, unexpected results can occur.

6.35.2 Cross reference

CWE: [None]
MISRA 2004: 6.1, 6.2, 6.3
JSF C++: 148, 183

6.35.3 Categorization

See clause 5.?.

6.35.4 Mechanism of failure

[Note: Mention the difference between name typing and structure typing. Mention coercion and casting.]

The *type* of a data object informs the compiler how values should be represented and which operations may be applied. The *type system* of a language is the set of rules used by the language to structure and organize its collection of types. Any attempt to manipulate data objects with inappropriate operations is a *type error*. A program is said to be *type safe* (or *type secure*) if it can be demonstrated that it has no type errors [2].

Every programming language has some sort of type system. A language is said to be *statically typed* if the type of every expression is known at compile time. The type system is said to be *strong* if it guarantees type safety and *weak* if it does not. There are strongly typed languages that are not statically typed because they enforce type safety with run time checks [2].

In practical terms, nearly every language falls short of being strongly typed (in an ideal sense) because of the inclusion of mechanisms to bypass type safety in particular circumstances. For that reason and because every language has a different type system, this description will focus on taking advantage of whatever features for type safety may be available in the chosen language.

Sometimes it is appropriate for a data value to be converted from one type to another *compatible* one. For example, consider the following program fragment, written in no specific language:

```
    float a;
    integer i;
    a := a + i;
```

The variable "i" is of integer type. It must be converted to the float type before it can be added to the data value. An implicit conversion, as shown, is called a *coercion*. If, on the other hand, the conversion must be explicit, e.g. "a := a + float(i)", then the conversion is called a *cast*.

Type *equivalence* is the strictest form of type compatibility; two types are equivalent if they are compatible without using coercion or casting. Type equivalence is usually characterized in terms of *name type equivalence*—two variables have the same type if they are declared in the same declaration or declarations that use the same type name—or *structure type equivalence*—two variables have the same type if they have identical structures. There are variations of these approaches and most languages use different combinations of them [1]. Therefore, a programmer skilled in one language may very well code inadvertent type errors when using a different language.

It is desirable for a program to be type safe because the application of operations to operands of an inappropriate type may produce unexpected results. (In addition, the presence of type errors can reduce the effectiveness of static analysis for other problems.) Searching for type errors is a valuable exercise because their presence often reveals design errors as well as coding errors. Many languages check for type errors—some at compile-time, others at run-time. Obviously, compile-time checking is more valuable because it can catch errors that are not executed by a particular set of test cases.

Making the most use of the type system of a language is useful in two ways. First, data conversions always bear the risk of changing the value. For example, a conversion from integer to float risks the loss of significant digits while the inverse conversion risks the loss of any fractional value. Second, a coder can use the type system to increase the probability of catching design errors or coding blunders. For example, the following Ada fragment declares two distinct floating point types:

```
    type Celsius is new Float;
    type Fahrenheit is new Float;
```

The declaration makes it impossible to add a value of type Celsius to a value of type Fahrenheit without explicit conversion.

6.35.5 Applicable language characteristics

This vulnerability description applies to most procedural programming languages.

6.35.6 Avoiding the vulnerability or mitigating its effects

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

- Take advantage of any facility offered by the programming language to declare distinct types and use any mechanism provided by the language processor and related tools to check for or enforce type compatibility.

- If possible, given the choice of language and processor, use available facilities to preclude or detect the occurrence of coercion. If it is not possible, use tooling and/or human review to assist in searching for coercions.
- Avoid casting data values except when there is no alternative. Document such occurrences so that the justification is made available to maintainers.
- Use the most restricted data type that suffices to accomplish the job. For example, use an enumeration type to select from a limited set of choices (e.g. a switch statement or the discriminant of a union type) rather than a more general type, such as integer. This will make it possible for tooling to check if all possible choices have been covered.
- Treat every compiler, tool, or run-time diagnostic concerning type compatibility as a serious issue. Do not resolve the problem by hacking the code with a cast; instead examine the underlying design to determine if the type error is a symptom of a deeper problem. Never ignore instances of coercion; if the conversion is necessary, convert it to a cast and document the rationale for use by maintainers.

6.35.7 Implications for standardization

It would be helpful if language specifiers used a common, uniform terminology to describe their type systems so that programmers experienced in other languages can reliably learn the type system of a language that is new to them.

It would be helpful if language implementers provided compiler switches or other tools to provide the highest possible degree of checking for type errors.

6.35.8 Bibliography

[1] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[2] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

6.36 CCB Enumerator issues

6.36.0 Status and history

2007-12-28 Edited by Stephen Michell

6.36.1 Description of application vulnerability

Enumerations are a finite list of named entities that contain a fixed mapping from a set of names to a set of integral values (called the representation) and an order between the members of the set. In some languages there are no other operations available except order, equality, first, last, previous, and next; in others the full underlying representation operators are available, such as integer "+" and "-" and bit-wise operations.

Most languages that provide enumeration types also provide mechanisms to set non-default representations. If these mechanisms do no enforce whole-type operations and check for conflicts then some members of the set may not be properly specified or may have the wrong maps. If the value-setting mechanisms are positional only, then there is a risk that improper counts or changes in relative order will result in an incorrect mapping.

For arrays indexed by enumerations with non-default representations, there is a risk of structures with holes, and if those indexes can be manipulated numerically, there is a risk of out-of-bound accesses of thise arrays.

Most of these errors can be readily detected by static analysis tools with appropriate coding standards, restrictions and annotations. Similarly mismatches in enumeration value specification can be detected statically. Without such rules, errors in the use of enumeration types are computationally hard to detect statically as well as being difficult to detect by human review.

6.36.2 Cross reference

MISRA 2004 - 9.1, 9.2, 9.3;
MISRA 32
JSF C++ Coding Standard 145;
Holzmann rule 6.

6.36.3 Categorization

See clause 5.?.

6.36.4 Mechanism of failure

As a program is developed and maintained the list of items in an enumeration often changes in three basic ways: New elements are added to the list; order between the members of the set often changes; and representation (the map of values of the items) change. Expressions that depend on the full set or specific relationships between elements of the set can create value errors which could result in wrong results or in unbounded behaviours if used as array indices.

Improperly mapped representations can result in some enumeration values being unreachable, or may create "holes" in the representation where undefinable values can be propagated.

If arrays are indexed by enumerations containing nondefault representations, some implementations may leave space for values that are unreachable using the enumeration, with a possibility of lost material or a way to pass information undetected (hidden channel).

When enumerators are set and initialized explicitly and the language permits incomplete initializers, then changes to the order of enumerators or the addition or deletion of enumerators can result in the wrong values being assigned or default values being assigned improperly. Subsequent indexing or switch/case structures can result in illegal accesses and possibly unbounded behaviours.

6.36.5 Applicable language Characteristics

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

Languages that provide named syntax for representation setting and coverage analysis can eliminate the order issues and incomplete coverage issues, as long as no "others" choices are used (e.g. The "when others =>" choice in Ada.

Languages that permit incomplete mappings between enumerator specification and value assignment, or that provide a positional-only mapping require additional static analysis tools and annotations to help identify the complete mapping of every literal to its value.

Languages that provide a trivial mapping to a type like integer require additional static analysis tools to prevent mixed type errors. They also cannot prevent illegal values from being placed into variables of such enumerator types; for example:

```
    enum Directions {back, forward, stop};
    Directions a = forward, b = forward, c = a+b;
```

In this example, c will have a value not defined by the enumeration, and any further use as that enumeration will lead to erroneous results.

Some languages provide no enumeration capability, leaving it to the programmer to define named constants to represent the values and ranges.

6.36.6 Avoiding the vulnerability or mitigating its effects

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

- Use static analysis that detect inappropriate use of enumerators, such as using them as integers or bit maps, and that detect enumeration definition expressions that are incomplete or incorrect. For languages with a complete enumeration abstraction this is the compiler.
- When positional notation is the only language-provided enumeration paradigm for assigning non-default values to enumerations, the use of comments to document the mapping between literals and their values helps humans and static analysis tools identify the intent and catch errors and changes.
- If the language permits partial assignment of representations to literals, always either initialize all items or none, and be explicit about any defaults assumed.
- When arrays are specified using enumerations as the index, only use enumeration types that have the default mapping.

Never perform numerical calculations on enumeration types

6.36.7 Implications for standardization

Languages that currently permit arithmetic and logical operations on enumeration types could provide a mechanism to ban such operations program-wide.

Languages that provide automatic defaults or that do not enforce static matching between enumerator definitions and initialization expressions could provide a mechanism to enforce such matching.

6.36.8 Bibliography

6.37 SYM Templates and generics

6.37.0 Status and history

2008-01-02: Updated by Clive Pygott
2007-12-12: Reviewed at OWGV meeting 7. Language-independent issues might include difficulties with human understanding, and difficulties in combining with other language features. On the other hand, it might turn out that sensible guidance is necessarily language-specific. It might be wise the review the entire document to find topics that should be revised to deal with their interaction with templates.
2007-10-15: Decided at OWGV meeting 6: Consider a description, SYM, related to templates and generics. Deal with JSF rules 101, 102, 103, 104, 105, 106.

6.37.1 Description of application vulnerability

Many languages provide a mechanism that allows objects and/or functions to be defined parameterized by type, and then instantiated for specific types. In C++ and related languages, these are referred to as “templates”, and in Ada and Java, “generics”. To avoid having to keep writing ‘templates/generics’, in this section these will simply be referred to collectively as generics.

Used well, generics can make code clearer, more predictable and easier to maintain. Used badly, they can have the reverse effect, making code difficult to review and maintain, leading to the possibility of program error.

6.37.2 Cross reference

JSF++: 100, 101, 102, 103, 104, 105
MISRA C++ 14-7-2 14-8-1 14-8-2

6.37.3 Categorization

See clause 5.?.

6.37.4 Mechanism of failure

The value of generics comes from having a single piece of code that supports some behaviour in a type independent manner. This simplifies development and maintenance of the code. It should also assist in the understanding of the code during review and maintenance, by providing the same behaviour for all types with which it is instantiated.

Problems arise when the use of a generic actually makes the code harder to understand during review and maintenance, by not providing consistent behaviour.

In most cases, the generic definition will have to make assumptions about the types it can legally be instantiated with. For example, a sort function requires that the elements to be sorted can be copied and compared. If these assumptions are not met, the result is likely to be a compiler error, for example if the sort function is instantiated with a user defined type that doesn't have a relational operator. Where 'misuse' of a generic leads to a compiler error, this can be regarded as a development issue, and not a software vulnerability.

Confusion, and hence potential vulnerability, can arise where the instantiated code is apparently illegal, but doesn't result in a compiler error. For example, a generic class defines a series of members, a subset of which relay on a particular property of the instantiation type (e.g. a generic container class with a sort member function, only the sort function relies on the instantiating type having a defined relational operator). In some languages, such as C++, if the generic is instantiated with a type that doesn't meet all the requirements but the program never subsequently makes use of the subset of members that rely on the property of the instantiating type, the code will compile and execute (e.g. the generic container is instantiated with a user defined class that doesn't define a relational operator, but the program never calls the sort member of this instantiation). When the code is reviewed the generic class will appear to reference a member of the instantiating type which doesn't exist.

Similar confusion can arise if the language permits specific elements of a generic to be explicitly defined, rather than using the common code, so that behaviour is not consistent for all instantiations. For example, for the same generic container class, the sort member normally sorts the elements of the container into ascending order. In languages such as C++, a 'special case' can be created for the instantiation of the generic with a particular type. For example, the sort member for a 'float' container may be explicitly defined to provide different behaviour, say sorting the elements into descending order. Specialization that doesn't affect the apparent behaviour of the instantiation is not an issue. Again, for C++, there are some irregularities in the semantics of arrays and pointers that can lead to the generic having different behaviour for different, but apparently very similar, types. In such cases, specialization can be used to enforce consistent behaviour.

6.37.5 Applicable language characteristics

This vulnerability applies to languages that permit definitions of objects or functions to be parameterized by type, for later instantiation with specific types, e.g.:

templates:	C++
generics:	Ada, Java

6.37.6 Avoiding the vulnerability or mitigating its effects

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

- document the properties of an instantiating type necessary for the generic to be valid
- if an instantiating type has the required properties, the whole of the generic should be valid, whether actually used in the program or not

- preferably avoid, but at least carefully document, any ‘special cases’ where the generic instantiated with a specific type doesn’t behave as it does for other types

6.37.7 Implications for standardization

6.37.8 Bibliography

6.38 LAV Initialization of variables

6.38.0 Status and history

2007-12-28 Initial write-up by Stephen Michell

6.38.1 Description of application vulnerability

All variables must contain a legal value that is a member of their type before the first time it is read. Reading a variable that has not been initialized with a legal value can cause unpredictable execution in the block that has visibility to the variable, and has the potential to export bad values to callers, or cause out of bounds memory accesses.

Uninitialized variable usage is often not detected until after testing and often when the code in question is delivered and in use, often because happenstance will provide it/them with adequate values (such as default data settings or accidental left-over values) until some other change exposes it the defect.

Variables that are declared during module construction (such as a class constructor, instantiation, or elaboration) may have alternate paths that can read values before they are set. This can happen in straight sequential code but is more prevalent when concurrency or co-routines are present, with the same impacts described above.

Another vulnerability occurs when compound objects are initialized incompletely, as can happen when objects are incrementally built, or fields are added under maintenance.

Depending on the compiler, linker, loader and runtime system some classes of objects may be preloaded with a known null or bad value, but systems should not rely on initialization (vice of static analysis) to catch initialization faults .

When possible and supported by the language, whole-structure initialization is preferable to field-by-field initialization statements, and named association is preferable to positional, as it facilitates human review and is less susceptible to failures under maintenance. For classes, the declaration and initialization may occur in separate modules. In such cases it must be possible to show that every field that needs an initial value receives that value, and to document ones that do not require initial values.

6.38.2 Cross reference

MISRA 9.1, 9.2 9.3
JSF C++ Coding Std 71, 143, 147

6.38.3 Categorization

See clause 5.?.

6.38.4 Mechanism of failure

Uninitialized objects may have illegal values, legal but wrong values, or legal and dangerous values. Wrong values could cause unbounded branches in conditionals or unbounded loop executions, or could simply cause wrong calculations and results.

There is a special case of pointers or access types. When such a type contains null values, a bound violation and likely hardware exception can result.; when such a type contains plausible but meaningless values, random data reads and writes can collect erroneous data or can destroy data that is in use by anoher part of the program; when such a type is an access to a subprogram with a plausible (but wrong) value, then either a bad instruction trap may occur or a transfer to an unknown code fragment can occur. All of these scenarios can result in unbounded behaviours.

Uninitialized variables are difficult to identify and use for attackers, but can be arbitrarily dangerous in safety situations.

6.38.5 Applicable Language Characteristics

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

- Some languages are defined such that all initialization must be constructed from sequential and possibly conditional operations, increasing the possibility that not all portions will be initialized.
- Some languages have elaboration time initialization and function invocation that can initialize objects as they are declared and before the first subprogram execution statement, permitting verifiable initialization before unit execution commences (when appropriate).

Some languages that have named assignments that can be used to build reviewable assignment structures that can be analyzed by the language processor for completeness. Languages with positional notation only can use comments and secondary tools to help show correct assignment.

6.38.6 Avoiding the vulnerability or mitigating its effects

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

The general problem of showing that all objects are initialized is intractable; hence developers must carefully structure programs to show that all variables are set before first read on every path throughout the subprogram.

The simplest method is to initialize each object at elaboration time, or immediately after subprogram execution commences and before any branches. If the subprogram must commence with conditional statements, then the programmer is responsible to show that every variable declared and not initialized earlier is initialized on each branch.

Applications can consider defining or reserving fields or portions of the object to only be set when initialized.

Where objects are visible from many modules, it is complex to determine where the first read occurs, and identify a module that must set the value before that read. When concurrency, interrupts and coroutines are present, it becomes especially imperative to identify where early initialization occurs and to show that the correct order is set via program structure, not by timing, OS precedence, or chance..

It should be possible to use static analysis tools to show that all objects are set before use in certain specific cases, but as the general problem is intractable, programmers should keep initialization algorithms simple so that they can be analyzed.

When declaring and initializing the object together, if the language does not statically match the declarative structure and the initialization structure, use static analysis tools to help detect any mismatches.

When setting compound objects, if the language provides mechanisms to set all components together, use those in preference to a sequence of initializations as this helps coverage analysis; otherwise use tools that perform such coverage analysis and document the initialization. Do not perform partial initializations unless there is no choice, and document any deviations from 100% initialization.

Where default assignment to multiple components are performed, explicit declaration of the component names and/or ranges helps static analysis and identification of component changes during maintenance.

6.38.7 Implications for standardization

Some languages have ways to determine if modules and regions are elaborated and initialized and to raise exceptions if this does not occur. Languages that do not may consider adding such capabilities.

Languages could consider setting aside fields in all objects to identify if initialization has occurred, especially for security and safety domains.

Languages that do not support whole-object initialization could consider adding this capability.

6.38.8 Bibliography

6.39 SAM Side-effects and order of evaluation

[For the convenience of reviewers, the applicable JSF C++ rule is quoted below:]

AV Rule 204: A single operation with side-effects shall only be used in the following contexts:

1. by itself
2. the right-hand side of an assignment
3. a condition
4. the only argument expression with a side-effect in a function call
5. condition of a loop
6. switch condition
7. single part of a chained operation.

Rationale: Readability

From JSF C++ Appendix (with examples)

AV Rule 204 attempts to prohibit side-effects in expressions that would be unclear, misleading, obscure, or would otherwise result in unspecified or undefined behavior. Consequently, an operation with side-effects will only be used in the following contexts:

Note: It is permissible for a side-effect to occur in conjunction with a constant expression. However, care should be taken so that additional side-effects are not "hidden" within the expression.

Note: Functions f(), g(), and h() have side-effects.

1. by itself

```
++i;              // Good
for (int32 i=0 ; i<max ; ++i)  // Good: includes the expression portion of a
                //              for statement
i++ - ++j;      // Bad: operation with side-effect doesn't occur by itself.
```

2. the right-hand side of an assignment

```
y = f(x);             // Good
y = ++x;              // Good: logically the same as y=f(x)
y = (-b + sqrt(b*b -4*a*c))/(2*a);   // Good: sqrt() does not have side-effect
y = f(x) + 1;            // Good: side-effect may occur with a constant
y = g(x) + h(z); // Bad: operation with side-effect doesn't occur by itself
              //          on rhs of assignment
k = i++ - ++j;        // Bad: same as above
y = f(x) + z;     // Bad: same as above
```

3. a condition

```
if (x.f(y))       // Good
if (int x = f(y))       // Good: this form is often employed with dynamic casts
```

```
          //             if (D* pd = dynamic_cast<D*> (pb)) {…}
if (++p == NULL) /// Good: side-effect may occur with a constant
if (i++ - --j)       // Bad: operation with side-effect doesn’t occur by itself
          //           in a condition
```

4. the only argument expression with a side-effect in a function call

```
f(g(z));             // Good
f(g(z),h(w));        // Bad: two argument expressions with side-effects
f(++i,++j);          // Bad: same as above
f(g(z), 3);          // Good: side-effect may occur with a constant
```

5. condition of a loop

```
while (f(x))         // Good
while(--x)        // Good
while((c=*p++) != -1)    // Bad: operation with side-effect doesn’t occur by itself
             //           in a loop condition
```

6. switch condition

```
switch (f(x))        // Good
switch (c = *p++)    // Bad: operation with side-effect doesn’t occur by itself
          //           in a switch condition
```

7. single part of a chained operation

```
x.f().g().h();           // Good
a + b * c;        // Good: (operator+() and operator*() are overloaded)
cout << x << y;      // Good
```

AV Rule 204.1

Since the order in which operators and subexpression are evaluated is unspecified, expressions must be written in a manner that produces the same value under any order the standard permits. . [Note from Tom: It isn’t just the value of the expression that must be the same; the values of all modified lvalues should also be the same, to achieve the goal of predictable behavior.]

`i = v[i++];       // Bad: unspecified behavior` **[Note from Tom: actually, it’s undefined behavior]**

`i = ++i + 1;      // Bad: unspecified behavior` **[Note from Tom: actually, it’s undefined behavior]**

`p->mem_func(*p++);   // Bad: unspecified behavior` **[Note from Tom: actually, it’s undefined behavior]**

[For the convenience of reviewers, I have paraphrased relevant rules from MISRA 2004:]

12.2 (req) The value of an expression shall be the same under any order of evaluation that the standard permits. [Note from Tom: It isn’t just the value of the expression that must be the same; the values of all modified lvalues should also be the same, to achieve the goal of predictable behavior.]

[For the convenience of reviewers, the applicable CERT/CC Guidelines are quoted below]
EXP30-C Do not depend on order of evaluation between sequence points
EXP35-C Do not access or modify the result of a function call after a subsequent sequence point

6.39.0 Status and history

NEEDS TO BE WRITTEN: Tom Plum
2008-01-21: Revised by Thomas Plum
2007-12-12: Reviewed at OWGV meeting 7: Mine material in JCW-071101 and N0108. Determine whether the order of initialization fits here, in LAV, or needs a distinct description.

2007-10-15: Decided at OWGV Meeting 6: We decide to write three new descriptions: operator precedence, JCW; associativity, MTW; order of evaluation, SAM. Deal with MISRA 2004 rules 12.1 and 12.2; JSF C++ rules 204, 213.

6.39.1 Description of application vulnerability

Some programming languages permit subexpressions to cause side-effects (such as assignment, increment, or decrement). For example, C and C++ permit such side-effects, and if, within one expression (such as "`i = v[i++]`"), two or more side-effects modify the same object, undefined behavior results (subject to certain restrictions that need not be recited here).

Some languages permit subexpressions to be computed in an unspecified ordering. If these subexpressions contain side-effects, then the value of the full expression can be dependent upon the order of evaluation. Furthermore, the objects that are modified by the side-effects can receive values that are dependent upon the order of evaluation.

If a program causes these unspecified or undefined behaviors, testing the program and seeing that it yields the expected results may give the false impression that the expression will always yield the correct result.

6.39.2 Cross reference

JSF C++ AV Rules 204, 204.1
MISRA 2004: 12.2
CERT/CC Guidelines: EXP30-C, EXP35-C

6.39.3 Categorization

See clause 5.?.

6.39.4 Mechanism of failure

When subexpressions with side effects are used within an expression, the unspecified order of evaluation can result in a program producing different results on different platforms, or even at different times on the same platform. For example, consider

```
a = f(b) + g(b);
```

where `f` and `g` both modify `b`. If `f(b)` is evaluated first, then the `b` used as a parameter to `g(b)` may be a different value than if `g(b)` is performed first. Likewise, if `g(b)` is performed first, `f(b)` may be called with a different value of `b`.

Other examples of unspecified order, or even undefined behavior, can be manifested, such as

```
a = f(i) + i++;
```

or

```
a[i++] = b[i++];
```

Parentheses around expressions can assist in removing ambiguity about grouping, but the issues regarding side-effects and order of evaluation remain; consider

```
j = i++ * i++;
```

where even if parentheses are placed around the `i++` subexpressions, undefined behavior still remains. (All examples above pertain to C and to C++.)

6.39.5 Applicable language characteristics

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

- Subexpressions with side effects can be used within an expression
- Subexpressions are computed in an unspecified ordering.

6.39.6 Avoiding the vulnerability or mitigating its effects

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

- Make use of one or more programming guidelines which (a) prohibit these unspecified or undefined behaviors, (b) can be enforced by static analysis, and (c) can be learned and understood by the relevant programmers.

6.39.7 Implications for standardization

In developing new or revised languages, give consideration to language restrictions which will eliminate or mitigate this vulnerability.

6.39.8 Bibliography

6.40 TEX Loop control variables

6.40.0 Status and history

2008-02-12, Initial version by Derek Jones
2007-12-12: Considered at OWGV meeting 7; Was mistakenly named TMP for a brief period.
2007-10-15: Decided at OWGV Meeting 6: Write a new description, TEX, about not messing with the control variable of a loop. MISRA 2004 rules 13.5, 13.6, 14.6; JSF C++ rules 198, 199, 200.

6.40.1 Description of application vulnerability

Many languages support a looping construct whose number of iterations is controlled by the value of a loop control variable. Looping constructs provide a method of specifying an initial value for this loop control variable, a test that terminates the loop and the quantity by which it should be decremented/incremented on each loop iteration.

In some languages it is possible to modify the value of the loop control variable within the body of the loop. Experience shows that such value modifications are sometimes overlooked by readers of the source code, resulting in faults being introduced.

6.40.2 Cross reference

MISRA-C:2004 rule 13.6
JSF C++ rule 201

6.40.3 Categorization

See clause 5.?. *<Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>*

6.40.4 Mechanism of failure

Readers of source code often make assumptions about what has been written. A common assumption is that a loop control variable is not modified in the body of its associated loop (such variables are not usually modified in the body of a loop). A reader of the source may incorrectly assume that a loop control variable is modified in the body of its loop and write (incorrect) code based on this assumption.

6.40.5 Applicable language characteristics

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

Languages that permit a loop control variable to be modified in the body of its associated loop (some languages (e.g., Ada) treat such usage as an erroneous construct and require translators to diagnose it).

6.40.6 Avoiding the vulnerability or mitigating its effects

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

Not modifying a loop control variable in the body of its associated loop body.

Some languages (e.g., C and C++) do not explicitly specify which of the variables appearing in a loop header is the loop control variable. Jones [?} and MISRA-C [?] have proposed algorithms for deducing which, if any, of these variables is the loop control variable in C (these algorithms could also be applied to other languages that support a C-like for-loop).

[Note: Do not under stand the [?} and [?] where these to be bib references?]

6.40.7 Implications for standardization

Whether or not loop control variables can be modified in the body of a loop is an existing language design decision and there is nothing new that this TR can suggest to language designers.

6.40.8 Bibliography

MISRA-C:2004 Guidelines for the use of the C language in critical systems

JOINT STRIKE FIGHTER AIR VEHICLE C++ CODING STANDARDS FOR THE SYSTEM DEVELOPMENT AND DEMONSTRATION PROGRAM, Lockheed Martin Corporation. Document Number 2RDU00001 Rev C, December 2005

Loops and their control variables: Discussion and proposed guidelines, Derek M. Jones, February 2006.

6.41 EWD Structured Programming

6.41.0 Status and history

2008-02-12, edited by Benito
2007-12-12, edited at OWGV meeting 7
2007-11-19, edited by Benito
2007-10-15, decided at OWGV meeting #6: "Write a new description, EWD about the use of structured programming that discusses `goto`, `continue` statement, `break` statement, single exit from a function. Discuss in terms of cost to analyzability and human understanding. Include `setjmp` and `longjmp`."

6.41.1 Description of application vulnerability

Programs that have a convoluted control structure are likely to be more difficult to be human readable, less understandable, harder to maintainable, more difficult to modify, harder to statically analyze, and more difficult to match the allocation and release of resources.

6.41.2 Cross reference

JSF: none
MISRA: 14.4 through 14.7 and 20.7

6.41.3 Categorization

See clause 5.?.

6.41.4 Mechanism of failure

- Memory or resource leaks
- Maintenance is error prone
- Validation of the design is difficult.
- Difficult to statically analyze.

6.41.5 Applicable language characteristics

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

- Languages that allow `goto` statements.
- Languages that allow leaving a loop without consideration for the loop control.
- Languages that allow local jumps (the `goto` statement).
- Languages that allow non-local jumps (`setjmp`/`longjmp` in the 'C' programming language).
- Languages that support multiple entry and exit points from a function, procedure, subroutine or method.

6.41.6 Avoiding the vulnerability or mitigating its effects

Use only those features of the programming language that enforces a logical structure on the program. The program flow follows a simple hierarchical model that employs looping constructs such as `for`, `repeat`, and `while`.

- Avoid using language features such as `goto.`
- Avoid using language features such as `continue` and `break` in the middle of loops.
- Avoid using language features that transfer control of the program flow via a jump.
- Avoid multiple exit points to a function/procedure/method/subroutine.
- Avoid multiple entry points to a function/procedure/method/subroutine.

6.41.7 Implications for standardization

6.41.8 Bibliography

Holtzmann-1

6.42 CSJ Passing parameters and return values

[For reference by reviewers, the relevant JSF rules are quoted below:
[AV Rule 69 A member function that does not affect the state of an object (its instance variables) will be declared const. Member functions should be const by default. Only when there is a clear, explicit reason should the const modifier on member functions be omitted.
[AV Rule 116 Small, concrete-type arguments (two or three words in size) should be passed by value if changes made to formal parameters should not be reflected in the calling function.
[AV Rule 117 Arguments should be passed by reference if NULL values are not possible.
[AV Rule 117.1 An object should be passed as const T& if the function should not change the value of the object.
[AV Rule 117.2 An object should be passed as T& if the function may change the value of the object.

[AV Rule 118.1 An object should be passed as const T* if its value should not be modified.
[AV Rule 118.2 An object should be passed as T* if its value may be modified.]

[For convenient reference by reviewers, I have paraphrased MISRA 2004 rules that may be appropriate:
[16.7 A pointer parameter in a function prototype should be declared as a point to const if the pointer is not used to modify the addressed object.
16.9 A function identifier shall only be used with either a preceding &, or with a parenthesized parameter list, which may be empty.]

6.42.0 Status and history

2007-12-18: Jim Moore, revised to deal with comments at OWGV meeting 7. Changes are marked using Track Changes.
2007-12-12: edited at OWGV meeting 7, see notes below. Also, cross-reference to Pygott contribution N0108, Order of Evaluation, and reassign JSF rule 111 to DCM.
2007-12-01: first draft by Jim Moore
2007-10-15: Decided at OWGV Meeting 6: Write a new description, CSJ, to deal with passing parameters and return values. Deal with passing by reference versus value; also with passing pointers. Distinguish mutable from non-mutable entities whenever possible.

6.42.1 Description of application vulnerability

Nearly every procedural language provides some method of process abstraction permitting decomposition of the flow of control into routines, functions, subprograms, or methods. (For the purpose of this description, the term subprogram will be used.) To have any effect on the computation, the subprogram must change data visible to the calling program. It can do this by changing the value of a non-local variable, changing the value of a parameter, or, in the case of a function, providing a return value. Because different languages use different mechanisms with different semantics for passing parameters, a programmer using an unfamiliar language may obtain unexpected results.

6.42.2 Cross reference

CERT: DCL33-C
CWE: [none]
MISRA 2004: 16.7, 16.9 [Added by Moore]
JSF C++: 69, 116, 117, 118

6.42.3 Categorization

See clause 5.?.

6.42.4 Mechanism of failure

(This description closely follows [2].) The mechanisms for parameter passing include: *call by reference*, *call by copy*, and *call by name*. The last is so specialized that it will not be treated in this description.

In call by reference, the calling program passes the addresses of the arguments to the called subprogram. When the subprogram references the corresponding formal parameter, it is actually sharing data with the calling program. If the subprogram changes a formal parameter, then the corresponding actual argument is also changed. If the actual argument is an expression or a constant, then the address of a temporary location is passed to the subprogram; this may be an error in some languages. Some languages may control changes to formal parameters based on labels such as `in`, `out`, or `inout`.

In call by copy, the called subprogram does not share data with the calling program. Instead, formal parameters act as local variables. Values are passed between the actual arguments and the formal parameters by copying. There are three cases to consider: *call by value* for `in` parameters; *call by result* for `out` parameters and function return values; and *call by value-result* for `inout` parameters. For call by value, the calling program evaluates the actual arguments and copies the result to the corresponding formal parameters which are then treated as local variables by the subprogram. For call by value, the values of the locals corresponding to formal parameters are copied to the corresponding actual arguments. For call by value-result, the values are copied in from the actual arguments at the beginning of the subprogram's execution and back out to the actual arguments at its termination.

The obvious disadvantage of call by copy is that extra copy operations are needed and execution time is required to produce the copies. Particularly if parameters represent sizable objects, such as large arrays, the cost of call by copy can be high. For this reason, many languages provide the call by reference mechanism. The disadvantage of call by reference is that the calling program cannot be assured that the subprogram hasn't changed data that was intended to be unchanged. For example, if an array is passed by reference to a subprogram intended to sum its elements, the subprogram could also change the values of one or more elements of the array. However, some languages enforce the subprogram's access to the shared data based on the labeling of actual arguments with modes—such as **`in`**, **`out`**, or **`inout`**.

A more difficult problem with call by reference is unintended aliasing. It is possible that the address of one actual argument is the same as another actual argument or that two arguments overlap in storage. A subprogram, assuming the two formal parameters to be distinct, may treat them inappropriately. For example, if one codes a subprogram to swap two values using the exclusive-or method, then a call to **`swap(x,x)`** will zero the value of x. Aliasing can also occur between arguments and non-local objects. For example, if a subprogram modifies a non-local object as a side-effect of its execution, referencing that object by a formal parameter will result in aliasing and, possibly, unintended results.

Some languages provide only simple mechanisms for passing data to subprograms, leaving it to the programmer to synthesize appropriate mechanisms. Often, the only available mechanism is to use call by copy to pass small scalar values or pointer values containing addresses of data structures. Of course, the latter amounts to using call by reference with no checking by the language processor. In such cases, subprograms can pass back pointers to anything whatsoever, including data that is corrupted or absent.

Some languages use call by copy for small objects, such as scalars, and call by reference for large objects, such as arrays. The choice of mechanism may even be implementation-defined. Because the two mechanisms produce different results in the presence of aliasing, it is very important to avoid aliasing.

An additional complication with subprograms occurs when one or more of the arguments are expressions. In such cases, the evaluation of one argument might have side-effects that result in a change to the value of another or unintended aliasing. Implementation choices regarding order of evaluation could affect the result of the computation. This particular problem is described in SAM.

6.42.5 Applicable language characteristics

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

- Procedural languages that provide mechanisms for defining subprograms where the data passes between the calling program and the subprogram via parameters and return values. This includes methods in many popular object-oriented languages.

6.42.6 Avoiding the vulnerability or mitigating its effects

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

- Use available mechanisms to label parameters as constants or with modes like **`in`**, **`out`**, or **`inout`**.
- When a choice of mechanisms is available, pass small simple objects using call by copy.
- When a choice of mechanisms is available and the computational cost of copying is tolerable, pass larger objects using call by copy.
- When the choice of language or the computational cost of copying forbids using call by copy, then take safeguards to prevent aliasing:
    - Minimize side-effects of subprograms on non-local objects; when side-effects are coded, ensure that the affected non-local objects are not passed as parameters using call by reference.
    - To avoid unintentional aliasing, avoid using expressions or functions as actual arguments; instead assign the result of the expression to a temporary local and pass the local.

  - Utilize tooling or other forms of analysis to ensure that non-obvious instances of aliasing are absent.

6.42.7 Implications for standardization

- Programming language specifications could provide labels—such as `in`, `out`, and `inout`—that control the subprogram's access to its formal parameters, and enforce the access.

6.42.8 Bibliography

[1] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008
[2] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

6.43 DCM Dangling references to stack frames

6.44.0 Status and history

2008-02-14: edited by Erhard Ploedereder: revised example, word polishing
2007-12-12: edited by OWGV meeting 7
2007-12-06: first version by Erhard Ploedereder
2007-10-15: Needs to be written.
2007-10-15, Decided at OWGV #6: We decide to write a new vulnerability, Pointer Arithmetic, RVG, for 17.1 thru 17.4. Don't do 17.5. We also want to create DCM to deal with dangling references to stack frames, 17.6. XYK deals with dangling pointers. Deal with MISRA 2004 rules 17.1, 17.2, 17.3, 17.4, 17.5, 17.6; JSF rule 175.

6.44.1 Description of application vulnerability

Many systems implementation languages allow treating the address of a local variable as a value stored in other variables. Examples are the application of the address operator in C or C++, or of the ‘Access or ‘Address attributes in Ada. In the C-family of languages, this facility is also used to model the call-by-reference mechanism by passing the address of the actual parameter by-value. An obvious safety requirement is that the stored address shall not be used after the lifetime of the local variable has expired. Technically, the stack frame, in which the local variable lived, has been popped and memory may have been reused for a subsequent call. Unfortunately the invalidity of the stored address is very difficult to decide. This situation can be described as a “dangling reference to the stack”. See also XYK “dangling references to the heap”.

6.44.2 Cross reference

JSF C++: 111
MISRA 2004: 17.6

6.44.3 Categorization

See clause 5

6.44.4 Mechanism of failure

The consequences of dangling references to the stack come in two flavors: a deterministically predictable flavor, which therefore can be exploited, and an intermittent, non-deterministic flavor, which is next to impossible to elicit during testing. The following code sample illustrates the two flavors; the behavior is not language-specific:

```
struct s {  … };
typedef struct s array_type[1000];
array_type* ptr;
```

```
array_type* F()
{
  struct s Arr[1000];
  ptr = &Arr;      // Risk of flavor 1;
  return &Arr;     // Risk of flavor 2;
}

…

  struct s secret;
  array_type* ptr2;
  ptr2 = F();
  secret = (*ptr2)[10];   // Fault of flavor 2

 …

  secret = (*ptr)[10];    // Fault of flavor 1
```

The risk of flavor 1 is the assignment of the address of Arr to a pointer variable that survives the lifetime of Arr. The fault is the subsequent use of the dangling reference to the stack, which references memory since altered by other calls and possibly validly owned by other routines. As part of a call-back, the fault allows systematic examination of portions of the stack contents without triggering an array-bounds-checking violation. Thus, this vulnerability is easily exploitable. As a fault, the effects can be most astounding, as memory gets corrupted by completely unrelated code portions. (A life-time check as part of pointer assignment can prevent the risk. In many cases, e.g., the situations above, the check is statically decidable by a compiler; however, for the general case, a dynamic check is needed to ensure that the copied pointer value lives no longer than the designated object.)

The risk of flavor 2 is an idiom “seen in the wild” to return the address of a local variable in order to avoid an expensive copy of a function result, as long as it is consumed before the next routine call occurs. The idiom is based on the ill-founded assumption that the stack will not be affected by anything until this next call is issued. The assumption is false, however, if an interrupt occurs and interrupt handling employs a strategy called “stack stealing”, i.e., using the current stack to satisfy its memory requirements. Thus, the value of Arr can be overwritten before it can be retrieved after the call on F. As this fault will only occur if the interrupt arrives after the call has returned but before the returned result is consumed, the fault is highly intermittent and next to impossible to (re-)create during testing. Thus, it is unlikely to be exploitable, but also exceedingly hard to find by testing. It can begin to occur after a completely unrelated interrupt handler has been coded or altered. Only static analysis can relatively easily detect the danger (unless the code combines it with risks of flavor 1). Some compilers issue warnings for this situation; such warnings need to be heeded.

6.44.5 Applicable language characteristics

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

- The address of a local entity (or formal parameter) of a routine can be obtained and stored in a variable or can be returned by this routine as a result; and
- no check is made that the lifetime of the variable receiving the address is no larger than the lifetime of the designated entity.

6.44.6 Avoiding the vulnerability or mitigating its effects

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

- Do not use the address of declared entities as storable, assignable or returnable value (except where idioms of the language make it unavoidable).
- Where unavoidable, ensure that the lifetime of the variable containing the address is completely enclosed by the lifetime of the designated object.
- Never return the address of a local variable as the result of a function call. (No excuses.)

6.44.7 Implications for standardization

Language designers can avoid the vulnerability or mitigate its ill effects in the following ways:

- Do not provide means to obtain the address of a declared entity as a storable value; or
- Define implicit checks to implement the assurance of enclosed lifetime expressed in 6.44.6. Note that, in many cases, the check is statically decidable, e.g., when the address of a local entity is taken as part of a return statement or expression.

6.44.8 Bibliography

6.45 GDL Recursion

[For the convenience of reviewers, I have paraphrased relevant rules from MISRA 2004: 16.2 Functions shall not call themselves, either directly or indirectly.]

6.45.0 Status and history

2007-12-17: Jim Moore: I edited this by accepting the changes marked in OWGV meeting 7.
2007-12-12: Edited by OWGV meeting 7
2007-12-07: Drafted by Jim Moore
2007-10-15: Decided at OWGV Meeting 6: Write a new description, GDL, suggesting that if recursion is used, then you have to deal with issues of termination and resource exhaustion.

6.45.1 Description of application vulnerability

Recursion is an elegant mathematical mechanism for defining the values of some functions. It is tempting to write code that mirrors the mathematics. However, the use of recursion in a computer can have a profound effect on the consumption of finite resources, leading to denial of service.

6.45.2 Cross reference

CWE:
MISRA 2004: 16.2
JSF C++:

6.45.3 Categorization

See clause 5.?.

6.45.4 Mechanism of failure

Mathematical recursion provides for the economical definition of some mathematical functions. However, economical definition and economical calculation are two different subjects. It is tempting to calculate the value of a recursive function using recursive subprograms because the expression in the programming language is straightforward and easy to understand. However, the impact on finite computing resources can be profound. Each invocation of a recursive subprogram may result in the creation of a new stack frame, complete with local variables. If stack space is limited (and it always is), then the calculation of some values will lead to an exhaustion of resources, that is, a denial of service.

In calculating the values of mathematical functions the use of recursion in a program is usually obvious, but this is not true in the general case. For example, finalization of a computing context after treating an error condition might result in recursion (e.g. attempting to "clean up" by closing a file after an error was encountered in closing the same file). Although such situations may have other problems, they typically do not result in exhaustion of resources but may otherwise result in a denial of service.

6.45.5 Applicable language characteristics

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

- Any language that permits the recursive invocation of subprograms.

6.45.6 Avoiding the vulnerability or mitigating its effects

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

- Converting recursive calculations to the corresponding iterative calculation. In principle, any recursive calculation can be remodeled as an iterative calculation which will have a much smaller impact on computing resources but which may be harder for a human to comprehend. The cost to human understanding must be weighed against the practical limits of computing resource.
- In cases where the depth of recursion can be shown to be statically bounded by a tolerable number, then recursion may be acceptable, but should be documented for the use of maintainers.

It should be noted that some languages or implementations provide special (more economical) treatment of a form of recursion known as *tail-recursion*. In this case, the impact on computing economy is minimized. When using such a language, tail recursion may be preferred to an iterative calculation.

6.45.7 Implications for standardization

[None]

6.45.8 Bibliography

[None]

6.46 NZN Returning error status

[For the convenience of reviewers, the applicable JSF C++ rule is quoted below:
[AV Rule 208 C++ exceptions **shall not** be used (i.e. *throw*, *catch* and *try* shall not be used.)
[Rationale: Tool support is not adequate at this time.]

[For the convenience of reviewers, I have paraphrased relevant rules from MISRA 2004:
[16.10 If a function returns error information, then that error information shall be tested.]

6.46.0 Status and history

OK: Jim Moore is responsible
2007-12-18: Jim Moore, minor editorial changes
2007-12-07: Drafted by Jim Moore
2007-10-15: Decided at OWGV Meeting 6: Write a new description, NZN, about returning error status. Some languages return codes that must be checked; others raise exceptions that must be handled. Deal with tool limitations related to exception handling; exceptions may not be statically analyzable.

6.46.1 Description of application vulnerability

Unpredicted error conditions--perhaps from hardware (such as an I/O device error), perhaps from software (such as heap exhaustion)—sometimes arise during the execution of code. Different programming languages provide a surprisingly wide variety of mechanisms to deal with such errors. The choice of a mechanism that doesn't match the programming language can lead to errors in the execution of the software or unexpected termination of the program. This could lead to a simple decrease in the robustness of a program or it could be exploited in a denial of service attack.

6.46.2 Cross reference

CWE: [None]
MISRA 2004: 16.10 [Added by Jim]
JSF C++: 208

6.46.3 Categorization

See clause 5.?.

6.46.4 Mechanism of failure

Even in the best-written programs, error conditions sometimes arise. Some errors occur because of defects in the software itself, but some result from external conditions in hardware, such as errors in I/O devices, or in the software system, such as exhaustion of heap space. If left untreated, the effect of the error might result in termination of the program or continuation of the program with incorrect results. To deal with the situation, designers of programming languages have equipped their languages with different mechanisms to detect and treat such errors. These mechanisms are typically intended to be used in specific programming idioms. However, the mechanisms differ among languages. A programmer expert in one language might mistakenly use an inappropriate idiom when programming in a different language with the result that some errors are left untreated, leading to termination or incorrect results. Attackers can exploit such weaknesses in denial of service attacks.

In general, languages make no distinction between dealing with programming errors (like an access to protected memory), unexpected hardware errors (like device error), expected but unusual conditions (like end of file), and even usual conditions that fail to provide the typical result (like an unsuccessful search). This description will use the term "error" to apply to all of the above. The description applies equally to error conditions that are detected via hardware mechanisms and error conditions that are detected via software during execution of a subprogram (such as an inappropriate parameter value).

6.46.5 Applicable language characteristics

Different programming languages provide remarkably different mechanisms for treating errors. In languages that provide a number of error detection and treatment mechanisms, it becomes a design issue to match the mechanism to the condition. This section will describe the mechanisms that are provided in widely used languages.

The simplest case is the set of languages that provide no special mechanism for the notification and treatment of unusual conditions. In such languages, error conditions are signaled by the value of an auxiliary status variable, often a subprogram parameter. C standard library functions use a variant of this approach; the error status is provided as the return value. Obviously, in such languages, it is imperative to check and act upon the status variable after every call to a subprogram that might provide an error indication. If error conditions can occur in an asynchronous manner, it is necessary to provide means to check for errors in a systematic and periodic manner.

Some languages, like Fortran, permit the passing of a label parameter to a subprogram or library routine. If an error is encountered, the subprogram returns to the indicated label rather than to the point at which it was called. Similarly some languages accept the name of a subprogram to be used to handle errors. In either case, it is imperative to provide labeled code or a subprogram to deal with all possible error situations.

The approaches described above have the disadvantage that error checking must be provided at every call to a subprogram. This can clutter the code immensely to deal with situations that may occur rarely. For this reason, some languages provide an exception mechanism that automatically transfers control when an error is encountered. This has the potential advantage of allowing error treatment to be factored into distinct error handlers, leaving the main execution path to deal with the usual results. The disadvantages, of course, are that the language design is complicated and the programmer must deal with the conceptually more complex problem of providing error handlers that are removed from the immediate context of a specific call to a subprogram. Furthermore, different languages provide exception handling mechanisms that differ in the manner in which various design issues are treated:

- How is the occurrence of an exception bound to a particular handler?
- What happens when no handler is local to an exception occurrence? Is the exception propagated in some manner or is it lost?
- What happens after an exception handler executes? Is control returned to the point before the call or after the call, or is the calling routine terminated in some way? If the calling routine is terminated, is there some provision for finalization, such as closing files or releasing resources?
- Are programmers permitted to define additional exceptions?
- Does the language provide default handlers for some exceptions or must the programmer explicitly provide for all of them?
- Can predefined exceptions be raised explicitly by a subprogram?
- Under what circumstances can error checking be disabled?

6.46.6 Avoiding the vulnerability or mitigating its effects

Given the variety of error handling mechanisms, it is difficult to write general guidelines. However, dealing with exception handlers can stress the capability of many static analysis tools and can, in some cases, reduce the effectiveness of their analysis. Therefore, for situations where the highest of reliability is required, the application should be designed so that exception handling is not used at all. In the more general case, exception handling mechanisms should be reserved for truly unexpected situations and other situations (possibly hardware arithmetic overflow) where no other mechanism is available. Situations which are merely unusual, like end of file, should be treated by explicit testing—either prior to the call which might raise the error or immediately afterward.

Checking error return values or auxiliary status variables following a call to a subprogram is mandatory unless it can be demonstrated that the error condition is impossible.

In dealing with languages where untreated exceptions can be lost (e.g. an exception that goes untreated within an Ada task), it is mandatory to deal with the exception in the local context before it is lost.

When execution within a particular context is abandoned due to an exception, it is important to finalize the context by closing open files, releasing resources and restoring any invariants associated with the context.

It is often not appropriate to repair an error condition and retry the operation. In such cases, one often treats a symptom but not the underlying problem. It is usually a better solution to finalize and terminate the current context and retreat to a context where the situation is known.

Error checking provided by the language, the software system, or the hardware should never be disabled in the absence of a conclusive analysis that the error condition is rendered impossible.

Because of the complexity of error handling, careful review of all error handling mechanisms is appropriate.

In applications with the highest requirements for reliability, defense-in-depth approaches are often appropriate, i.e. checking and handling errors thought to be impossible.

6.46.7 Implications for standardization

[None]

6.46.8 Bibliography

[1] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[2] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

6.47 REU Termination strategy

6.47.0 Status and history

2008-01-10 Edited by Larry Wagoner
NEEDS TO BE REVISED: Larry Wagoner. Tom Plum will provide additional ideas. Also, Dan Nagle. Tom will describe "run time constraint handler" from 24731-1.
2007-12-13: Considered by OWGV, meeting 7: Try to keep this one in Clause 6, rather than 7. Discuss issues involved in clean-up to terminate the program or selected parts of the program.
2007-11-27: Drafted in part by Larry Wagoner
2007-10-15 Decided at OWGV meeting 6: Write a new description, REU, that discusses abnormal termination of programs, fail-soft, fail-hard, fail-safe. You need to have a strategy and select appropriate language features and library components. Deal with MISRA 2004 rule 20.11.

6.47.1 Description of application vulnerability

Expectations that a system will be dependable are based on the confidence that the system will operate as expected and not fail in normal use. The dependability of a system and its fault tolerance can be measured through the component part's reliability, availability, safety and security. Reliability is the ability of a system or component to perform its required functions under stated conditions for a specified period of time [IEEE 1990 glossary]. Availability is how timely and reliable the system is to its intended users. Both of these factors matter highly in systems used for safety and security. In spite of the best intentions, systems will encounter a failure, either from internally poorly written software or external forces such as power outages/variations, floods, or other natural disasters. The reaction to a fault can affect the performance of a system and in particular, the safety and security of the system and its users.

When a fault is detected, there are many ways in which a system can react. The quickest and most noticeable way is to fail hard, also known as fail fast or fail stop. The reaction to a detected fault is to immediately halt the system. Alternatively, the reaction to a detected fault could be to fail soft. The system would keep working with the faults present, but the performance of the system would be degraded. Systems used in a high availability environment such as telephone switching centers, e-commerce, etc. would likely use a fail soft approach. What is actually done in a fail soft approach can vary depending on whether the system is used for safety critical or security critical purposes. For fail safe systems, such as flight controllers, traffic signals, or medical monitoring systems, there would be no effort to meet normal operational requirements, but rather to limit the damage or danger caused by the fault. A system that fails securely, such as cryptologic systems, would maintain maximum security when a fault is detected, possibly through a denial of service.

6.47.2 Cross reference

MISRA 2004: 20.11

6.47.3 Categorization

See clause 5.?.

6.47.4 Mechanism of failure

The reaction to a fault in a system can depend on the criticality of the part in which the fault originates. When a program consists of several tasks, the tasks each may be critical, or not. If a task is critical, it may or may not be restartable by the rest of the program. Ideally, a task which detects a fault within itself should be able to halt leaving its resources available for use by the rest of the program, halt clearing away its resources, or halt the entire program. The latency of any such communication, and whether other tasks can ignore such a communication, should be clearly specified. Having inconsistent reactions to a fault, such as the fault reaction to a crypto fault, can potentially be a vulnerability.

6.47.5 Applicable language characteristics

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

6.47.6 Avoiding the vulnerability or mitigating its effects

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

- A strategy for fault handling should be decided. Consistency in fault handling should be the same with respect to critically similar parts.
- A multi-tiered approach of fault prevention, fault detection and fault reaction should be used.
- System-defined components that assist in uniformity of fault handling should be used when available. For one example, designing a "runtime constraint handler" (as described in ISO/IEC TR 24731-1) permits the application to intercept various erroneous situations and perform one consistent response, such as flushing a previous transaction and re-starting at the next one.
    - When there are multiple tasks, a fault-handling policy should be specified whereby a task may
    - halt, and keep its resources available for other tasks (perhaps permitting restarting of the faulting task)
    - halt, and remove its resources (perhaps to allow other tasks to use the resources so freed, or to allow a recreation of the task)
    - halt, and signal the rest of the program to likewise halt.

6.47.7 Implications for standardization

6.47.8 Bibliography

6.48 BVQ Unspecified Functionality

6.48.0 Status and history

2008-01-02: Updated by Clive Pygott
2007-12-13: OWGV Meeting 7: created this vulnerability to be based largely on Clive's N0108.

6.48.1 Description of application vulnerability

‘Unspecified functionality’ is code that may be executed, but whose behaviour does not contribute to the requirements of the application. Whilst this may be no more than an amusing ‘Easter Egg’, like the flight simulator in Microsoft’s Excel 97, it does raise questions about the level of control of the development process.

In a security-critical environment particularly, could the developer of an application have included a ‘trap-door’ to allow illegitimate access to the system on which it is eventually executed, irrespective of whether the application has obvious security requirements or not?

6.48.2 Cross reference

XYQ: Dead and Deactivated code. Dead and deactivated code is unnecessary code that exists in the binary but is never executed, whilst unspecified functionality is unnecessary code (as far as the requirements of the program are concerned) that exists in the binary and which may be executed.

6.48.3 Categorization

See clause 5.?.

6.48.4 Mechanism of failure

Unspecified functionality is not a software vulnerability per se, but more a development issue. In some cases, unspecified functionality may be added by a developer without the knowledge of the development organization (for example an aircraft auto-pilot that was programmed to fly around the developer's home town). In other cases, typically Easter Eggs, the functionality is unspecified as far as the user is concerned (nobody buys a spreadsheet expecting to find it includes a flight simulator), but is specified by the development organization. In effect they only reveal a subset of the program's behaviour to the users.

In the first case, one would expect a well managed development environment to discover the additional functionality during validation and verification. In the second case, the user is relying on the supplier not to release harmful code.

In effect, a program's requirements are 'the program should behave in the following manner …. and do nothing else'. The 'and do nothing else' clause is often not explicitly stated, and can be difficult to demonstrate.

6.48.5 Applicable language characteristics

This vulnerability description is intended to be applicable to all languages.

6.48.6 Avoiding the vulnerability or mitigating its effects

End user's can avoid the vulnerability or mitigate its ill effects in the following ways:

- programs that are to be used in critical applications should come from a developer with a recognized and audited development process. For example: ISO9001 or CMM.
- the development process should generate documentation showing traceability from source code to requirements, in effect answering 'why is this unit of code in this program?'. Where unspecified functionality is there for a legitimate reason (e.g. diagnostics required for developer maintenance or enhancement), the documentation should also record this. It is not unreasonable for customers of bespoke critical code to ask to see such traceability as part of their acceptance of the application

6.48.7 Implications for standardization

6.48.8 Bibliography

7. Application Vulnerabilities

7.1 RST Injection

7.1.0 Status and history

2007-08-04, Edited by Benito
2007-07-30, Created by Larry Wagoner
Combined:
XYU-070720-sql-injection-hibernate.doc
XYV-070720-php-file-inclusion.doc
XZC-070720-equivalent-special-element-injection.doc
XZD-070720-os-command-injection.doc
XZE-070720-injection.doc
XZF-070720-delimiter.doc
XZG-070720-server-side-injection.doc
XZJ-070720-common-special-element-manipulations.doc
into RST-070730-injection.doc.

7.1.1 Description of application vulnerability

(XYU) Using Hibernate to execute a dynamic SQL statement built with user input can allow an attacker to modify the statement's meaning or to execute arbitrary SQL commands.

(XYV) A PHP product uses "require" or "include" statements, or equivalent statements, that use attacker-controlled data to identify code or HTML to be directly processed by the PHP interpreter before inclusion in the script.

(XZC) The software allows the injection of special elements that are non-typical but equivalent to typical special elements with control implications into the dataplane. This frequently occurs when the product has protected itself against special element injection.

(XZD) Command injection problems are a subset of injection problem, in which the process can be tricked into calling external processes of an attacker's choice through the injection of command syntax into the data plane.

(XZE) Injection problems span a wide range of instantiations. The basic form of this weakness involves the software allowing injection of control-plane data into the data-plane in order to alter the control flow of the process.

(XZF) Line or section delimiters injected into an application can be used to compromise a system. as data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions that result in an attack.

(XZG) The software allows inputs to be fed directly into an output file that is later processed as code, e.g. a library file or template. A web product allows the injection of sequences that cause the server to treat as server-side includes.

(XZJ) Multiple leading/internal/trailing special elements injected into an application through input can be used to compromise a system. As data is parsed, improperly handled multiple leading special elements may cause the process to take unexpected actions that result in an attack.

7.1.2 Cross reference

CWE:
76. Equivalent Special Element Injection
78. OS Command Injection

90. LDAP Injection
91. XML Injection (aka Blind Xpath injection)
92. Custom Special Character Injection
95. Direct Dynamic Code Evaluation ('Eval Injection')
97. Server-Side Includes (SSI) Injection
98 PHP File Inclusion
99. Resource Injection
144. Line Delimiter
145. Section Delimiter
161. Multiple Leading Special Elements
163. Multiple Trailing Special Elements
165. Multiple Internal Special Elements
166. Missing Special Element
167. Extra Special Element
168. Inconsistent Special Elements
564. SQL Injection: Hibernate

7.1.3 Categorization

See clause 5.?. <*Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*>

7.1.4 Mechanism of failure

(XYU) SQL injection attacks are another instantiation of injection attack, in which SQL commands are injected into data-plane input in order to effect the execution of predefined SQL commands. Since SQL databases generally hold sensitive data, loss of confidentiality is a frequent problem with SQL injection vulnerabilities.

If poor SQL commands are used to check user names and passwords, it may be possible to connect to a system as another user with no previous knowledge of the password. If authorization information is held in a SQL database, it may be possible to change this information through the successful exploitation of a SQL injection vulnerability. Just as it may be possible to read sensitive information, it is also possible to make changes or even delete this information with a SQL injection attack.

(XYV) This is frequently a functional consequence of other weaknesses. It is usually multi-factor with other factors, although not all inclusion bugs involve assumed-immutable data. Direct request weaknesses frequently play a role. This can also overlap directory traversal in local inclusion problems.

(XZC) Many injection attacks involve the disclosure of important information -- in terms of both data sensitivity and usefulness in further exploitation. In some cases injectable code controls authentication; this may lead to a remote vulnerability. Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code. Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing. Often the actions performed by injected control code are not logged.

(XZD) A software system that accepts and executes input in the form of operating system commands (e.g. `system()`, `exec()`, `open()`) could allow an attacker with lesser privileges than the target software to execute commands with the elevated privileges of the executing process.

Command injection is a common problem with wrapper programs. Often, parts of the command to be run are controllable by the end user. If a malicious user injects a character (such as a semi-colon) that delimits the end of one command and the beginning of another, he may then be able to insert an entirely new and unrelated command to do whatever he pleases. The most effective way to deter such an attack is to ensure that the input provided by the user adheres to strict rules as to what characters are acceptable. As always, white-list style checking is far preferable to black-list style checking.

Dynamically generating operating system commands that include user input as parameters can lead to command injection attacks. An attacker can insert operating system commands or modifiers in the user input that can cause the request to behave in an unsafe manner. Such vulnerabilities can be very dangerous and lead to data and system compromise. If no validation of the parameter to the exec command exists, an attacker can execute any command on the system the application has the privilege to access.

Command injection vulnerabilities take two forms: an attacker can change the command that the program executes (the attacker explicitly controls what the command is); or an attacker can change the environment in which the command executes (the attacker implicitly controls what the command means). In this case we are primarily concerned with the first scenario, in which an attacker explicitly controls the command that is executed. Command injection vulnerabilities of this type occur when:

- Data enters the application from an untrusted source.
- The data is part of a string that is executed as a command by the application.
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

(XZE) Injection problems encompass a wide variety of issues -- all mitigated in very different ways. For this reason, the most effective way to discuss these weaknesses is to note the distinct features which classify them as injection weaknesses. The most important issue to note is that all injection problems share one thing in common -- they allow for the injection of control plane data into the user controlled data plane. This means that the execution of the process may be altered by sending code in through legitimate data channels, using no other mechanism. While buffer overflows and many other flaws involve the use of some further issue to gain execution, injection problems need only for the data to be parsed. The most classic instantiations of this category of weakness are SQL injection and format string vulnerabilities.

Many injection attacks involve the disclosure of important information in terms of both data sensitivity and usefulness in further exploitation. In some cases injectable code controls authentication, this may lead to a remote vulnerability.

Injection attacks are characterized by the ability to significantly change the flow of a given process, and in some cases, to the execution of arbitrary code.

Data injection attacks lead to loss of data integrity in nearly all cases as the control-plane data injected is always incidental to data recall or writing. Often the actions performed by injected control code are not logged.

Eval injection occurs when the software allows inputs to be fed directly into a function (e.g. "eval") that dynamically evaluates and executes the input as code, usually in the same interpreted language that the product uses. Eval injection is prevalent in handler/dispatch procedures that might want to invoke a large number of functions, or set a large number of variables.

A PHP file inclusion occurs when a PHP product uses "require" or "include" statements, or equivalent statements, that use attacker-controlled data to identify code or HTML to be directly processed by the PHP interpreter before inclusion in the script.

A resource injection issue occurs when the following two conditions are met:

- An attacker can specify the identifier used to access a system resource. For example, an attacker might be able to specify part of the name of a file to be opened or a port number to be used.
- By specifying the resource, the attacker gains a capability that would not otherwise be permitted.

For example, the program may give the attacker the ability to overwrite the specified file, run with a configuration controlled by the attacker, or transmit sensitive information to a third-party server. Note: Resource injection that involves resources stored on the file system goes by the name path manipulation and is reported in separate category. See the path manipulation description for further details of this vulnerability. Allowing user input to control resource identifiers may enable an attacker to access or modify otherwise protected system resources.

(XZF) Line or section delimiters injected into an application can be used to compromise a system. as data is parsed, an injected/absent/malformed delimiter may cause the process to take unexpected actions that result in an attack. One example of a section delimiter is the boundary string in a multipart MIME message. In many cases, doubled line delimiters can serve as a section delimiter.

(XZG) This can be resultant from XSS/HTML injection because the same special characters can be involved. However, this is server-side code execution, not client-side.

(XZJ) The software does not respond properly when an expected special element (character or reserved word) is missing, an extra unexpected special element (character or reserved word) is used or an inconsistency exists between two or more special characters or reserved words, e.g. if paired characters appear in the wrong order, or if the special characters are not properly nested.

7.1.5 Avoiding the vulnerability or mitigating its effects

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

- (XYU) A non-SQL style database which is not subject to this flaw may be chosen.
- Follow the principle of least privilege when creating user accounts to a SQL database. Users should only have the minimum privileges necessary to use their account. If the requirements of the system indicate that a user can read and modify their own data, then limit their privileges so they cannot read/write others' data.
- Duplicate any filtering done on the client-side on the server side.
- Implement SQL strings using prepared statements that bind variables. Prepared statements that do not bind variables can be vulnerable to attack.
- Use vigorous white-list style checking on any user input that may be used in a SQL command. Rather than escape meta-characters, it is safest to disallow them entirely since the later use of data that have been entered in the database may neglect to escape meta-characters before use.
- Narrowly define the set of safe characters based on the expected value of the parameter in the request.
- (XZC) As so many possible implementations of this weakness exist, it is best to simply be aware of the weakness and work to ensure that all control characters entered in data are subject to black-list style parsing.
- (XZD) Assign permissions to the software system that prevents the user from accessing/opening privileged files.
- (XZE) A language can be chosen which is not subject to these issues.
- As so many possible implementations of this weaknes exist, it is best to simply be aware of the weakness and work to ensure that all control characters entered in data are subject to black-list style parsing. Assume all input is malicious. Use an appropriate combination of black lists and white lists to ensure only valid and expected input is processed by the system.
- To avert eval injections, refractor your code so that it does not need to use `eval()` at all.
- (XZF) Developers should anticipate that delimiters and special elements will be injected/removed/manipulated in the input vectors of their software system. Use an appropriate combination of black lists and white lists to ensure only valid, expected and appropriate input is processed by the system.
- (XZG) Assume all input is malicious. Use an appropriate combination of black lists and white lists to ensure only valid and expected input is processed by the system.

7.1.6 Implications for standardization

7.1.7 Bibliography

7.2 EWR Path Traversal

7.2.0 Status and history

PENDING
2007-08-05, Edited by Benito
2007-07-13, Created by Larry Wagoner
Combined

XYA-070720-relative-path-traversal.doc
XYB-070720-absolute-path-traversal.doc
XYC-070720-path-link-problems.doc
XYD-070720-windows-path-link-problems.doc
into EWR-070730-path-traversal

7.2.1 Description of application vulnerability

The software can construct a path that contains relative traversal sequences such as ".."

The software can construct a path that contains absolute path sequences such as "/path/here."

Attackers running software in a particular directory so that the hard link or symbolic link used by the software accesses a file that the attacker has control over may be able to escalate their privilege level to that of the running process.

Attackers running software in a particular directory so that the hard link or symbolic link used by the software accesses a file that the attacker has control over may be able to escalate their privilege level to that of the running process.

7.2.2 Cross reference

CWE:

24. Path Issue - dot dot slash - '../filedir'
25. Path Issue - leading dot dot slash - '/../filedir'
26. Path Issue - leading directory dot dot slash - '/dir
27. Path Issue - directory doubled dot dot slash - 'directory/../../filename'
28. Path Issue - dot dot backslash - '..\filename'
29. Path Issue - leading dot dot backslash - '\..\filename'
30. Path Issue - leading directory dot dot backslash - '\directory\..\filename'
31. Path Issue - directory doubled dot dot backslash - 'directory\..\..\filename'
32. Path Issue - triple dot - '...'
33. Path Issue - multiple dot - '....'
34. Path Issue - doubled dot dot slash - '....//'
35. Path Issue - doubled triple dot slash - '.../...//'
37. Path Issue - slash absolute path - /absolute/pathname/here
38. Path Issue - backslash absolute path - \absolute\pathname\here
39. Path Issue - drive letter or Windows volume - 'C:dirname'
40. Path Issue - Windows UNC share - '\\UNC\share\name\'
61. UNIX symbolic link (symlink) following
62. UNIX hard link
64. Windows shortcut following (.LNK)
65. Windows hard link

7.2.3 Categorization

See clause 5.?.

7.2.4 Mechanism of failure

A software system that accepts input in the form of: '..\filename', '\..\filename', '/directory/../filename', 'directory/../../filename', '..\filename', '\..\filename', '\directory\..\filename', 'directory\..\..\filename', '...', '....' (multiple dots), '....//', or '.../...//' without appropriate validation can allow an attacker to traverse the file system to access an arbitrary file. Note that '..' is ignored if the current working directory is the root directory. Some of these input forms can be used to cause problems for systems that strip out '..' from input in an attempt to remove relative path traversal.

A software system that accepts input in the form of '/absolute/pathname/here' or '\absolute\pathname\here' without appropriate validation can allow an attacker to traverse the file system to unintended locations or access arbitrary files. An attacker can inject a drive letter or Windows volume letter ('C:dirname') into a software system to potentially redirect access to an unintended location or arbitrary file.

A software system that accepts input in the form of a backslash absolute path () without appropriate validation can allow an attacker to traverse the file system to unintended locations or access arbitrary files.

An attacker can inject a Windows UNC share ('\\UNC\share\name') into a software system to potentially redirect access to an unintended location or arbitrary file.

A software system that allows UNIX symbolic links (symlink) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The symbolic link can permit an attacker to read/write/corrupt a file that they originally did not have permissions to access.

Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if he/she can replace a file used by a privileged program with a hard link to a sensitive file (e.g. `etc/passwd`). When the process opens the file, the attacker can assume the privileges of that process.

A software system that allows Windows shortcuts (.LNK) as part of paths whether in internal code or through user input can allow an attacker to spoof the symbolic link and traverse the file system to unintended locations or access arbitrary files. The shortcut (file with the .lnk extension) can permit an attacker to read/write a file that they originally did not have permissions to access.

Failure for a system to check for hard links can result in vulnerability to different types of attacks. For example, an attacker can escalate their privileges if an he/she can replace a file used by a privileged program with a hard link to a sensitive file (e.g. etc/`passwd`). When the process opens the file, the attacker can assume the privileges of that process or possibly prevent a program from accurately processing data in a software system.

7.2.5 Avoiding the vulnerability or mitigating its effects

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

- Assume all input is malicious. Attackers can insert paths into input vectors and traverse the file system.
- Use an appropriate combination of black lists and white lists to ensure only valid and expected input is processed by the system.
- Warning: if you attempt to cleanse your data, then do so that the end result is not in the form that can be dangerous. A sanitizing mechanism can remove characters such as '.' and ';' which may be required fir some exploits. An attacker can try to fool the sanitizing mechanism into "cleaning" data into a dangerous form. Suppose the attacker injects a '.' inside a filename (e.g. "sensi.tiveFile") and the sanitizing mechanism removes the character resulting in the valid filename, "sensitiveFile". If the input data are now assumed to be safe, then the file may be compromised.
- Files can often be identified by other attributes in addition to the file name, for example, by comparing file ownership or creation time. Information regarding a file that has been created and closed can be stored and then used later to validate the identity of the file when it is reopened. Comparing multiple attributes of the file improves the likelihood that the file is the expected one.
- Follow the principle of least privilege when assigning access rights to files.
- Denying access to a file can prevent an attacker from replacing that file with a link to a sensitive file.
- Ensure good compartmentalization in the system to provide protected areas that can be trusted.

- When two or more users, or a group of users, have write permission to a directory, the potential for sharing and deception is far greater than it is for shared access to a few files. The vulnerabilities that result from malicious restructuring via hard and symbolic links suggest that it is best to avoid shared directories.
- Securely creating temporary files in a shared directory is error prone and dependent on the version of the runtime library used, the operating system, and the file system. Code that works for a locally mounted file system, for example, may be vulnerable when used with a remotely mounted file system.
- [The mitigation should be centered on converting relative paths into absolute paths and then verifying that the resulting absolute path makes sense with respect to the configuration and rights or permissions. This may include checking "whitelists" and "blacklists", authorized super user status, access control lists, etc.]

7.2.6 Implications for standardization

7.2.7 Bibliography

7.3 XYP Hard-coded Password

7.3.0 Status and history

Pending
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.3.1 Description of application vulnerability

Hard coded passwords may compromise system security in a way that cannot be easily remedied. It is never a good idea to hardcode a password. Not only does hardcoding a password allow all of the project's developers to view the password, it also makes fixing the problem extremely difficult. Once the code is in production, the password cannot be changed without patching the software. If the account protected by the password is compromised, the owners of the system will be forced to choose between security and availability.

7.3.2 Cross reference

CWE:
259. Hard-coded Password

7.3.3 Categorization

See clause 5.?.

7.3.4 Mechanism of failure

The use of a hard-coded password has many negative implications -- the most significant of these being a failure of authentication measures under certain circumstances. On many systems, a default administration account exists which is set to a simple default password which is hard-coded into the program or device. This hard-coded password is the same for each device or system of this type and often is not changed or disabled by end users. If a malicious user comes across a device of this kind, it is a simple matter of looking up the default password (which is freely available and public on the Internet) and logging in with complete access. In systems which authenticate with a back-end service, hard-coded passwords within closed source or drop-in solution systems require that the back-end service use a password which can be easily discovered. Client-side systems with hard-coded passwords propose even more of a threat, since the extraction of a password from a binary is exceedingly simple. If hard-coded passwords are used, it is almost certain that malicious users will gain access through the account in question.

7.3.5 Avoiding the vulnerability or mitigating its effects

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

- Rather than hard code a default username and password for first time logins, utilize a "first login" mode which requires the user to enter a unique strong password.
- For front-end to back-end connections, there are three solutions that may be used.
  - Use of generated passwords which are changed automatically and must be entered at given time intervals by a system administrator. These passwords will be held in memory and only be valid for the time intervals.
  - The passwords used should be limited at the back end to only performing actions valid to for the front end, as opposed to having full access.
  - The messages sent should be tagged and checksummed with time sensitive values so as to prevent replay style attacks.

7.3.6 Implications for standardization

7.3.7 Bibliography

7.4 XYS Executing or Loading Untrusted Code

7.4.0 Status and History

PENDING
2007-08-05, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.4.1 Description of application vulnerability

Executing commands or loading libraries from an untrusted source or in an untrusted environment can cause an application to execute malicious commands (and payloads) on behalf of an attacker.

7.4.2 Cross reference

CWE:
114. Process Control

7.4.3 Categorization

See clause 5.?.

7.4.4 Mechanism of failure

Process control vulnerabilities take two forms:

- An attacker can change the command that the program executes so that the attacker explicitly controls what the command is;
- An attacker can change the environment in which the command executes so that the attacker implicitly controls what the command means.

Considering only the first scenario, the possibility that an attacker may be able to control the command that is executed, process control vulnerabilities occur when:

- Data enters the application from an untrusted source.
- The data is used as or as part of a string representing a command that is executed by the application.
- By executing the command, the application gives an attacker a privilege or capability that the attacker would not otherwise have.

7.4.5 Avoiding the vulnerability or mitigating its effects

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

- Libraries that are loaded should be well understood and come from a trusted source. The application can execute code contained in the native libraries, which often contain calls that are susceptible to other security problems, such as buffer overflows or command injection.

- All native libraries should be validated to determine if the application requires the use of the library. It is very difficult to determine what these native libraries actually do, and the potential for malicious code is high. In addition, the potential for an inadvertent mistake in these native libraries is also high, as many are written in C or C++ and may be susceptible to buffer overflow or race condition problems.

- To help prevent buffer overflow attacks, validate all input to native calls for content and length.

- If the native library does not come from a trusted source, review the source code of the library. The library should be built from the reviewed source before using it.

7.4.6 Implications for standardization

7.4.7 Bibliography

7.5 XYM Insufficiently Protected Credentials

7.5.0 Status and History

Pending
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.5.1 Description of application vulnerability

This weakness occurs when the application transmits or stores authentication credentials and uses an insecure method that is susceptible to unauthorized interception and/or retrieval.

7.5.2 Cross reference

CWE:
256. Plaintext Storage
257. Storing Passwords in a Recoverable Format

7.5.3 Categorization

See clause 5.?. *<Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>*

7.5.4 Mechanism of failure

Storing a password in plaintext may result in a system compromise. Password management issues occur when a password is stored in plaintext in an application's properties or configuration file. A programmer can attempt to remedy the password management problem by obscuring the password with an encoding function, such as base 64 encoding, but this effort does not adequately protect the password. Storing a plaintext password in a configuration file allows anyone who can read the file access to the password-protected resource. Developers sometimes believe that they cannot defend the application from someone who has access to the configuration, but this attitude makes an attacker's job easier. Good password management guidelines require that a password never be stored in plaintext.

The storage of passwords in a recoverable format makes them subject to password reuse attacks by malicious users. If a system administrator can recover the password directly or use a brute force search on the information available to him, he can use the password on other accounts.

The use of recoverable passwords significantly increases the chance that passwords will be used maliciously. In fact, it should be noted that recoverable encrypted passwords provide no significant benefit over plain-text passwords since they are subject not only to reuse by malicious attackers but also by malicious insiders.

7.5.5 Avoiding the vulnerability or mitigating its effects

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

- Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:
- Avoid storing passwords in easily accessible locations.
- Never store a password in plaintext.
- Ensure that strong, non-reversible encryption is used to protect stored passwords.
- Consider storing cryptographic hashes of passwords as an alternative to storing in plaintext.

7.5.6 Implications for standardization

7.5.7 Bibliography

7.6 XYT Cross-site Scripting

7.6.0 Status and History

2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.6.1 Description of application vulnerability

Cross-site scripting (XSS) weakness occurs when dynamically generated web pages display input, such as login information, that is not properly validated, allowing an attacker to embed malicious scripts into the generated page and then execute the script on the machine of any user that views the site. If successful, Cross-site scripting vulnerabilities can be exploited to manipulate or steal cookies, create requests that can be mistaken for those of a valid user, compromise confidential information, or execute malicious code on the end user systems for a variety of nefarious purposes.

7.6.2 Cross reference

CWE:

80. Basic XSS
81. XSS in error pages
82. Script in IMG tags
83. XSS using Script in Attributes
84. XSS using Script Via Encoded URI Schemes
85. Doubled character XSS manipulators, e.g. '`<<script`'
86. Invalid Character in Identifiers
87. Alternate XSS syntax

7.6.3 Categorization

See clause 5.?.

7.6.4 Mechanism of failure

Cross-site scripting (XSS) vulnerabilities occur when an attacker uses a web application to send malicious code, generally JavaScript, to a different end user. When a web application uses input from a user in the output it generates without filtering it, an attacker can insert an attack in that input and the web application sends the attack to other users. The end user trusts the web application, and the attacks exploit that trust to do things that would not normally be allowed. Attackers frequently use a variety of methods to encode the malicious portion of the tag, such as using Unicode, so the request looks less suspicious to the user.

XSS attacks can generally be categorized into two categories: stored and reflected. Stored attacks are those where the injected code is permanently stored on the target servers in a database, message forum, visitor log, and so forth. Reflected attacks are those where the injected code takes another route to the victim, such as in an email message, or on some other server. When a user is tricked into clicking a link or submitting a form, the injected code travels to the vulnerable web server, which reflects the attack back to the user's browser. The browser then executes the code because it came from a 'trusted' server. For a reflected XSS attack to work, the victim must submit the attack to the server. This is still a very dangerous attack given the number of possible ways to trick a victim into submitting such a malicious request, including clicking a link on a malicious Web site, in an email, or in an inner-office posting.

XSS flaws are very likely in web applications, as they require a great deal of developer discipline to avoid them in most applications. It is relatively easy for an attacker to find XSS vulnerabilities. Some of these vulnerabilities can be found using scanners, and some exist in older web application servers. The consequence of an XSS attack is the same regardless of whether it is stored or reflected.

The difference is in how the payload arrives at the server. XSS can cause a variety of problems for the end user that range in severity from an annoyance to complete account compromise. The most severe XSS attacks involve disclosure of the user's session cookie, which allows an attacker to hijack the user's session and take over their account. Other damaging attacks include the disclosure of end user files, installation of Trojan horse programs, redirecting the user to some other page or site, and modifying presentation of content.

Cross-site scripting (XSS) vulnerabilities occur when:
1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious code.

The malicious content sent to the web browser often takes the form of a segment of JavaScript, but may also include HTML, Flash or any other type of code that the browser may execute. The variety of attacks based on XSS is almost limitless, but they commonly include transmitting private data like cookies or other session information to the attacker, redirecting the victim to web content controlled by the attacker, or performing other malicious operations on the user's machine under the guise of the vulnerable site.

Cross-site scripting attacks can occur wherever an untrusted user has the ability to publish content to a trusted web site. Typically, a malicious user will craft a client-side script, which — when parsed by a web browser — performs some activity (such as sending all site cookies to a given E–mail address). If the input is unchecked, this script will be loaded and run by each user visiting the web site. Since the site requesting to run the script has access to the cookies in question, the malicious script does also. There are several other possible attacks, such as running "Active X" controls (under Microsoft Internet Explorer) from sites that a user perceives as trustworthy; cookie theft is however by far the most common. All of these attacks are easily prevented by ensuring that no script tags — or for good measure, HTML tags at all — are allowed in data to be posted publicly.

Specific instances of XSS are:
'Basic' XSS involves a complete lack of cleansing of any special characters, including the most fundamental XSS elements such as "<", ">", and "&".

A web developer displays input on an error page (e.g. a customized 403 Forbidden page). If an attacker can influence a victim to view/request a web page that causes an error, then the attack may be successful.

A Web application that trusts input in the form of HTML IMG tags is potentially vulnerable to XSS attacks. Attackers can embed XSS exploits into the values for IMG attributes (e.g. SRC) that is streamed and then executed in a victim's browser. Note that when the page is loaded into a user's browsers, the exploit will automatically execute.

The software does not filter "javascript:" or other URI's from dangerous attributes within tags, such as onmouseover, onload, onerror, or style.

The web application fails to filter input for executable script disguised with URI encodings.

The web application fails to filter input for executable script disguised using doubling of the involved characters.

The software does not strip out invalid characters in the middle of tag names, schemes, and other identifiers, which are still rendered by some web browsers that ignore the characters.

The software fails to filter alternate script syntax provided by the attacker.

Cross-site scripting attacks may occur anywhere that possibly malicious users are allowed to post unregulated material to a trusted web site for the consumption of other valid users. The most common example can be found in bulletin-board web sites which provide web based mailing list-style functionality. The most common attack performed with cross-site scripting involves the disclosure of information stored in user cookies. In some circumstances it may be possible to run arbitrary code on a victim's computer when cross-site scripting is combined with other flaws.

7.6.5 Avoiding the vulnerability or mitigating its effects

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

- Carefully check each input parameter against a rigorous positive specification (white list) defining the specific characters and format allowed.
- All input should be sanitized, not just parameters that the user is supposed to specify, but all data in the request, including hidden fields, cookies, headers, the URL itself, and so forth.
- A common mistake that leads to continuing XSS vulnerabilities is to validate only fields that are expected to be redisplayed by the site.
- Data is frequently encountered from the request that is reflected by the application server or the application that the development team did not anticipate. Also, a field that is not currently reflected may be used by a future developer. Therefore, validating ALL parts of the HTTP request is recommended.

7.6.6 Implications for standardization

7.6.7 Bibliography

7.7 XYN Privilege Management

7.7.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.7.1 Description of application vulnerability

Failure to adhere to the principle of least privilege amplifies the risk posed by other vulnerabilities.

7.7.2 Cross reference

CWE:
250. Often Misused: Privilege Management

7.7.3 Categorization

See clause 5.?. <*Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*>

7.7.4 Mechanism of failure

This vulnerability type refers to cases in which an application grants greater access rights than necessary. Depending on the level of access granted, this may allow a user to access confidential information. For example, programs that run with root privileges have caused innumerable Unix security disasters. It is imperative that you carefully review privileged programs for all kinds of security problems, but it is equally important that privileged programs drop back to an unprivileged state as quickly as possible in order to limit the amount of damage that an overlooked vulnerability might be able to cause. Privilege management functions can behave in some less-than-obvious ways, and they have different quirks on different platforms. These inconsistencies are particularly pronounced if you are transitioning from one non-root user to another. Signal handlers and spawned processes run at the privilege of the owning process, so if a process is running as root when a signal fires or a sub-process is executed, the signal handler or sub-process will operate with root privileges. An attacker may be able to leverage these elevated privileges to do further damage. To grant the minimum access level necessary, first identify the different permissions that an application or user of that application will need to perform their actions, such as file read and write permissions, network socket permissions, and so forth. Then explicitly allow those actions while denying all else.

7.7.5 Avoiding the vulnerability or mitigating its effects

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

Very carefully manage the setting, management and handling of privileges. Explicitly manage trust zones in the software.

Follow the principle of least privilege when assigning access rights to entities in a software system.

7.7.6 Implications for standardization

7.7.7 Bibliography

7.8 XYO Privilege Sandbox Issues

7.8.0 Status and history

Pending
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.8.1 Description of application vulnerability

A variety of vulnerabilities occur with improper handling, assignment, or management of privileges. These are especially present in sandbox environments, although it could be argued that any privilege problem occurs within the context of some sort of sandbox.

7.8.2 Cross reference

CWE:
266. Incorrect Privilege Assignment
267. Unsafe Privilege
268. Privilege Chaining
269. Privilege Management Error
270. Privilege Context Switching Error
272. Least Privilege Violation
273. Failure to Check Whether Privileges were Dropped Successfully
274. Insufficient Privileges
276. Insecure Default Permissions

7.8.3 Categorization

See clause 5.?. <*Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*>

7.8.4 Mechanism of failure

The failure to drop system privileges when it is reasonable to do so is not an application vulnerability by itself. It does, however, serve to significantly increase the severity of other vulnerabilities. According to the principle of least privilege, access should be allowed only when it is absolutely necessary to the function of a given system, and only for the minimal necessary amount of time. Any further allowance of privilege widens the window of time during which a successful exploitation of the system will provide an attacker with that same privilege.

There are many situations that could lead to a mechanism of failure. A product could incorrectly assign a privilege to a particular entity. A particular privilege, role, capability, or right could be used to perform unsafe actions that were not intended, even when it is assigned to the correct entity. (Note that there are two separate sub-categories here: privilege incorrectly allows entities to perform certain actions; and the object is incorrectly accessible to entities with a given privilege.) Two distinct privileges, roles, capabilities, or rights could be combined in a way that allows an entity to perform unsafe actions that would not be allowed without that combination. The software may not properly manage privileges while it is switching between different contexts that cross privilege boundaries. A product may not properly track, modify, record, or reset privileges. In some contexts, a system executing with elevated permissions will hand off a process/file/etc. to another process/user. If the privileges of an entity are not reduced, then elevated privileges are spread throughout a system and possibly to an attacker. The software may not properly handle the situation in which it has insufficient privileges to perform an operation. A program, upon installation, may set insecure permissions for an object.

7.8.5 Avoiding the vulnerability or mitigating its effects

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

- The principle of least privilege when assigning access rights to entities in a software system should be followed. The setting, management and handling of privileges should be managed very carefully. Upon changing security privileges, one should ensure that the change was successful.
- Consider following the principle of separation of privilege. Require multiple conditions to be met before permitting access to a system resource.
- Trust zones in the software should be explicity managed. If at all possible, limit the allowance of system privilege to small, simple sections of code that may be called atomically.
- As soon as possible after acquiring elevated privilege to call a privileged function such as chroot(), the program should drop root privilege and return to the privilege level of the invoking user.
- In newer Windows implementations, make sure that the process token has the SeImpersonatePrivilege.

7.8.6 Implications for standardization

7.8.7 Bibliography

7.9 XZO Authentication Logic Error

7.9.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.9.1 Description of application vulnerability

The software does not properly ensure that the user has proven their identity.

7.9.2 Cross reference

CWE:
288. Authentication Bypass by Alternate Path/Channel
289. Authentication Bypass by Alternate Name
290. Authentication Bypass by Spoofing
294. Authentication Bypass by Replay
301. Reflection Attack in an Authentication Protocol
302. Authentication Bypass by Assumed-Immutable Data
303. Authentication Logic Error
305. Authentication Bypass by Primary Weakness

7.9.3 Categorization

See clause 5.?. *<Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>*

7.9.4 Mechanism of failure

Authentication bypass by alternate path or channel occurs when a product requires authentication, but the product has an alternate path or channel that does not require authentication. Note that this is often seen in web applications that assume that access to a particular CGI program can only be obtained through a "front" screen, but this problem is not just in web apps.

Authentication bypass by alternate name occurs when the software performs authentication based on the name of the resource being accessed, but there are multiple names for the resource, and not all names are checked.

Authentication bypass by capture-replay occurs when it is possible for a malicious user to sniff network traffic and bypass authentication by replaying it to the server in question to the same effect as the original message (or with minor changes). Messages sent with a capture-relay attack allow access to resources which are not otherwise accessible without proper authentication. Capture-replay attacks are common and can be difficult to defeat without cryptography. They are a subset of network injection attacks that rely listening in on previously sent valid commands, then changing them slightly if necessary and resending the same commands to the server. Since any attacker who can listen to traffic can see sequence numbers, it is necessary to sign messages with some kind of cryptography to ensure that sequence numbers are not simply doctored along with content.

Reflection attacks capitalize on mutual authentication schemes in order to trick the target into revealing the secret shared between it and another valid user. In a basic mutual-authentication scheme, a secret is known to both the valid user and the server; this allows them to authenticate. In order that they may verify this shared secret without sending it plainly over the wire, they utilize a Diffie-Hellman-style scheme in which they each pick a value, then request the hash of that value as keyed by the shared secret. In a reflection attack, the attacker claims to be a valid user and requests the hash of a random value from the server. When the server returns this value and requests its own value to be hashed, the attacker opens another connection to the server. This time, the hash requested by the attacker is the value which the server requested in the first connection. When the server returns this hashed value, it is used in the first connection, authenticating the attacker successfully as the impersonated valid user.

Authentication bypass by assumed-immutable data occurs when the authentication scheme or implementation uses key data elements that are assumed to be immutable, but can be controlled or modified by the attacker, e.g. if a web application relies on a cookie "`Authenticated=1`"

Authentication logic error occurs when the authentication techniques do not follow the algorithms that define them exactly and so authentication can be jeopardized. For instance, a malformed or improper implementation of an algorithm can weaken the authorization technique.

An authentication bypass by primary weakness occurs when the authentication algorithm is sound, but the implemented mechanism can be bypassed as the result of a separate weakness that is primary to the authentication error.

7.9.5 Avoiding the vulnerability or mitigating its effects

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

- Funnel all access through a single choke point to simplify how users can access a resource. For every access, perform a check to determine if the user has permissions to access the resource. Avoid making decisions based on names of resources (e.g. files) if those resources can have alternate names.
- Canonicalize the name to match that of the file system's representation of the name. This can sometimes be achieved with an available API (e.g. in Win32 the `GetFullPathName` function).

- Utilize some sequence or time stamping functionality along with a checksum which takes this into account in order to ensure that messages can be parsed only once.
- Use different keys for the initiator and responder or of a different type of challenge for the initiator and responder.
- Assume all input is malicious. Use an appropriate combination of black lists and white lists to ensure only valid and expected input is processed by the system. For example, valid input may be in the form of an absolute pathname(s). You can also limit pathnames to exist on selected drives, have the format specified to include only separator characters (forward or backward slashes) and alphanumeric characters, and follow a naming convention such as having a maximum of 32 characters followed by a '.' and ending with specified extensions.

7.9.6 Implications for standardization

7.9.7 Bibliography

7.10 XZX Memory Locking

7.10.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.10.1 Description of application vulnerability

Sensitive data stored in memory that was not locked or that has been improperly locked may be written to swap files on disk by the virtual memory manager.

7.10.2 Cross reference

CWE:
591. Memory Locking

7.10.3 Categorization

See clause 5.?. <Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>

7.10.4 Mechanism of failure

Sensitive data that is written to a swap file may be exposed.

7.10.5 Avoiding the vulnerability or mitigating its effects

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

- Identify data that needs to be protected from swapping and choose platform-appropriate protection mechanisms.
- Check return values to ensure locking operations are successful.

- On Windows systems the VirtualLock function can lock a page of memory to ensure that it will remain present in memory and not be swapped to disk. However, on older versions of Windows, such as 95, 98, or Me, the `VirtualLock()` function is only a stub and provides no protection. On POSIX systems the `mlock()` call ensures that a page will stay resident in memory but does not guarantee that the page will not appear in the swap. Therefore, it is unsuitable for use as a protection mechanism for sensitive data. Some platforms, in particular Linux, do make the guarantee that the page will not be swapped, but this is non-standard and is not portable. Calls to `mlock()` also require supervisor privilege. Return values for both of these calls must be checked to ensure that the lock operation was actually successful.

7.10.6 Implications for standardization

[Note: Should POSIX and other API standards should provide the functionality.]

7.10.7 Bibliography

7.11 XZP Resource Exhaustion

7.11.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.11.1 Description of application vulnerability

The application is susceptible to generating and/or accepting an excessive amount of requests that could potentially exhaust limited resources, such as memory, file system storage, database connection pool entries, or CPU. This can ultimately lead to a denial of service that could prevent valid users from accessing the application.

7.11.2 Cross reference

CWE:
400. Resource Exhaustion (file descriptor, disk space, sockets,...)

7.11.3 Categorization

See clause 5.?.

7.11.4 Mechanism of failure

There are two primary failures associated with resource exhaustion. The most common result of resource exhaustion is denial of service. In some cases it may be possible to force a system to "fail open" in the event of resource exhaustion.

Resource exhaustion issues are generally understood but are far more difficult to successfully prevent. Taking advantage of various entry points, an attacker could craft a wide variety of requests that would cause the site to consume resources. Database queries that take a long time to process are good DoS targets. An attacker would only have to write a few lines of Perl code to generate enough traffic to exceed the site's ability to keep up. This would effectively prevent authorized users from using the site at all.

Resources can be exploited simply by ensuring that the target machine must do much more work and consume more resources in order to service a request than the attacker must do to initiate a request. Prevention of these attacks requires either that the target system either recognizes the attack and denies that user further access for a given amount of time or uniformly throttles all requests in order to make it more difficult to consume resources more quickly than they can again be freed. The first of these solutions is an issue in itself though, since it may allow attackers to prevent the use of the system by a particular valid user. If the attacker impersonates the valid user, he may be able to prevent the user from accessing the server in question. The second solution is simply difficult to effectively institute and even when properly done, it does not provide a full solution. It simply makes the attack require more resources on the part of the attacker.

The final concern that must be discussed about issues of resource exhaustion is that of systems which "fail open." This means that in the event of resource consumption, the system fails in such a way that the state of the system — and possibly the security functionality of the system — is compromised. A prime example of this can be found in old switches that were vulnerable to "macof" attacks (so named for a tool developed by Dugsong). These attacks flooded a switch with random IP and MAC address combinations, therefore exhausting the switch's cache, which held the information of which port corresponded to which MAC addresses. Once this cache was exhausted, the switch would fail in an insecure way and would begin to act simply as a hub, broadcasting all traffic on all ports and allowing for basic sniffing attacks.

7.11.5 Avoiding the vulnerability or mitigating its effects

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

- Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:
- Implement throttling mechanisms into the system architecture. The best protection is to limit the amount of resources that an unauthorized user can cause to be expended. A strong authentication and access control model will help prevent such attacks from occurring in the first place. The login application should be protected against DoS attacks as much as possible. Limiting the database access, perhaps by caching result sets, can help minimize the resources expended. To further limit the potential for a DoS attack, consider tracking the rate of requests received from users and blocking requests that exceed a defined rate threshold.
- Other ways to avoid the vulnerability are to ensure that protocols have specific limits of scale placed on them, ensure that all failures in resource allocation place the system into a safe posture and to fail safely when resource exhaustion occurs.

7.11.6 Implications for standardization

7.11.7 Bibliography

7.12 XZQ Unquoted Search Path or Element

7.12.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.12.1 Description of application vulnerability

Strings injected into a software system that are not quoted can permit an attacker to execute arbitrary commands.

7.12.2 Cross reference

CWE:

428. Unquoted Search Path or Element

7.12.3 Categorization

See clause 5.?.

7.12.4 Mechanism of failure

The mechanism of failure stems from missing quoting of strings injected into a software system. By allowing whitespaces in identifiers, an attacker could potentially execute arbitrary commands. This vulnerability covers "`C:\Program Files`" and space-in-search-path issues. Theoretically this could apply to other operating systems besides Windows, especially those that make it easy for spaces to be in files or folders.

7.12.5 Avoiding the vulnerability or mitigating its effects

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

- Software should quote the input data that can be potentially executed on a system.

7.12.6 Implications for standardization

7.12.7 Bibliography

7.13 XZL Discrepancy Information Leak

7.13.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.13.1 Description of application vulnerability

A discrepancy information leak is an information leak in which the product behaves differently, or sends different responses, in a way that reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.13.2 Cross reference

CWE:
204. Response Discrepancy Information Leak
206. Internal Behavioral Inconsistency Information Leak
207. External Behavorial Inconsistency Information Leak
208. Timing Discrepancy Information Leak

7.13.3 Categorization

See clause 5.?. *<Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.>*

7.13.4 Mechanism of failure

A response discrepancy information leak occurs when the product sends different messages in direct response to an attacker's request, in a way that allows the attacker to learn about the inner state of the product. The leaks can be inadvertent (bug) or intentional (design).

A behavioural discrepancy information leak occurs when the product's actions indicate important differences based on (1) the internal state of the product or (2) differences from other products in the same class. Attacks such as OS fingerprinting rely heavily on both behavioral and response discrepancies. An internal behavioural inconsistency information leak is the situation where two separate operations in a product cause the product to behave differently in a way that is observable to an attacker and reveals security-relevant information about the internal state of the product, such as whether a particular operation was successful or not. An external behavioural inconsistency information leak is the situation where the software behaves differently than other products like it, in a way that is observable to an attacker and reveals security-relevant information about which product is being used, or its operating state.

A timing discrepancy information leak occurs when two separate operations in a product require different amounts of time to complete, in a way that is observable to an attacker and reveals security-relevant information about the state of the product, such as whether a particular operation was successful or not.

7.13.5 Avoiding the vulnerability or mitigating its effects

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

- Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:
- Compartmentalize your system to have "safe" areas where trust boundaries can be unambiguously drawn. Do not allow sensitive data to go outside of the trust boundary and always be careful when interfacing with a compartment outside of the safe area.

7.13.6 Implications for standardization

7.13.7 Bibliography

7.14 XZN Missing or Inconsistent Access Control

7.14.0 Status and history

PENDING
2007-08-04, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.14.1 Description of application vulnerability

The software does not perform access control checks in a consistent manner across all potential execution paths.

7.14.2 Cross reference

CWE:
285. Missing or Inconsistent Access Control

7.14.3 Categorization

See clause 5.?.

7.14.4 Mechanism of failure

For web applications, attackers can issue a request directly to a page (URL) that they may not be authorized to access. If the access control policy is not consistently enforced on every page restricted to authorized users, then an attacker could gain access to and possibly corrupt these resources.

7.14.5 Avoiding the vulnerability or mitigating its effects

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

- For web applications, make sure that the access control mechanism is enforced correctly at the server side on every page. Users should not be able to access any information that they are not authorized for by simply requesting direct access to that page. Ensure that all pages containing sensitive information are not cached, and that all such pages restrict access to requests that are accompanied by an active and authenticated session token associated with a user who has the required permissions to access that page.

7.14.6 Implications for standardization

7.14.7 Bibliography

7.15 XZS Missing Required Cryptographic Step

7.15.0 Status and history

PENDING
2007-08-03, Edited by Benito
2007-07-30, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.15.1 Description of application vulnerability

Cryptographic implementations should follow the algorithms that define them exactly otherwise encryption can be faulty.

7.15.2 Cross reference

CWE:
325. Missing Required Cryptographic Step

7.15.3 Categorization

See clause 5.?.

7.15.4 Mechanism of failure

Not following the algorithms that define cryptographic implementations exactly can lead to weak encryption.

7.15.5 Avoiding the vulnerability or mitigating its effects

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

- Implement cryptographic algorithms precisely.

7.15.6 Implications for standardization

[Note: This should be added to programming language libraries.]

7.15.7 Bibliography

7.16 XZR Improperly Verified Signature

7.16.0 Status and history

PENDING
2007-08-03, Edited by Benito
2007-07-27, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.16.1 Description of application vulnerability

The software does not verify, or improperly verifies, the cryptographic signature for data.

7.16.2 Cross reference

CWE:
347. Improperly Verified Signature

7.16.3 Categorization

See clause 5.?.

7.16.4 Mechanism of failure

7.16.5 Avoiding the vulnerability or mitigating its effects

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

7.16.6 Implications for standardization

7.16.7 Bibliography

7.17 XZK Sensitive Information Uncleared Before Use

7.17.0 Status and history

PENDING
2007-08-10, Edited by Benito
2007-08-08, Edited by Larry Wagoner
2007-07-20, Edited by Jim Moore
2007-07-13, Edited by Larry Wagoner

7.17.1 Description of application vulnerability

The software does not fully clear previously used information in a data structure, file, or other resource, before making that resource available to another party that did not have access to the original information.

7.17.2 Cross reference

CWE:
226. Sensitive Information Uncleared Before Use

7.17.3 Categorization

See clause 5.?.

7.17.4 Mechanism of failure

This typically involves memory in which the new data are not as long as the old data, which leaves portions of the old data still available ("memory disclosure"). However, equivalent errors can occur in other situations where the length of data is variable but the associated data structure is not. This can overlap with cryptographic errors and cross-boundary cleansing info leaks.

Dynamic memory managers are not required to clear freed memory and generally do not because of the additional runtime overhead. Furthermore, dynamic memory managers are free to reallocate this same memory. As a result, it is possible to accidentally leak sensitive information if it is not cleared before calling a function that frees dynamic memory. Programmers should not and can not rely on memory being cleared during allocation.

7.17.5 Avoiding the vulnerability or mitigating its effects

To prevent information leakage, sensitive information must be cleared from dynamically allocated buffers before they are freed.

7.17.6 Implications for standardization

Library functions and or language features that provide the function to clear the buffers.

7.17.7 Bibliography

1

Annex A
(informative)

Guideline Recommendation Factors

A. Guideline Recommendation Factors

A.1 Factors that need to be covered in a proposed guideline recommendation

These are needed because circumstances might change, for instance:

- Changes to language definition.
- Changes to translator behavior.
- Developer training.
- More effective recommendation discovered.

A.1.1 Expected cost of following a guideline

How to evaluate likely costs.

A.1.2 Expected benefit from following a guideline

How to evaluate likely benefits.

A.2 Language definition

Which language definition to use. For instance, an ISO/IEC Standard, Industry standard, a particular implementation.

Position on use of extensions.

A.3 Measurements of language usage

Occurrences of applicable language constructs in software written for the target market.

How often do the constructs addressed by each guideline recommendation occur.

A.4 Level of expertise.

How much expertise, and in what areas, are the people using the language assumed to have?

Is use of the alternative constructs less likely to result in faults?

A.5 Intended purpose of guidelines

For instance: How the listed guidelines cover the requirements specified in a safety related standard.

A.6 Constructs whose behaviour can very

The different ways in which language definitions specify behaviour that is allowed to vary between implementations and how to go about documenting these cases.

A.7 Example guideline proposal template

A.7.1 Coding Guideline

Anticipated benefit of adhering to guideline

- Cost of moving to a new translator reduced.
- Probability of a fault introduced when new version of translator used reduced.
- Probability of developer making a mistake is reduced.
- Developer mistakes more likely to be detected during development.
- Reduction of future maintenance costs.

Annex B
(informative)
Guideline Selection Process

B. Guideline Selection Process

It is possible to claim that any language construct can be misunderstood by a developer and lead to a failure to predict program behavior. A cost/benefit analysis of each proposed guideline is the solution adopted by this technical report.

The selection process has been based on evidence that the use of a language construct leads to unintended behavior (i.e., a cost) and that the proposed guideline increases the likelihood that the behavior is as intended (i.e., a benefit). The following is a list of the major source of evidence on the use of a language construct and the faults resulting from that use:

- a list of language constructs having undefined, implementation defined, or unspecified behaviours,
- measurements of existing source code. This usage information has included the number of occurrences of uses of the construct and the contexts in which it occurs,
- measurement of faults experienced in existing code,
- measurements of developer knowledge and performance behaviour.

The following are some of the issues that were considered when framing guidelines:

- An attempt was made to be generic to particular kinds of language constructs (i.e., language independent), rather than being language specific.
- Preference was given to wording that is capable of being checked by automated tools.
- Known algorithms for performing various kinds of source code analysis and the properties of those algorithms (i.e., their complexity and running time).

B.1 Cost/Benefit Analysis

The fact that a coding construct is known to be a source of failure to predict correct behavior is not in itself a reason to recommend against its use. Unless the desired algorithmic functionality can be implemented using an alternative construct whose use has more predictable behavior, then there is no benefit in recommending against the use of the original construct.

While the cost/benefit of some guidelines may always come down in favor of them being adhered to (e.g., don't access a variable before it is given a value), the situation may be less clear cut for other guidelines. Providing a summary of the background analysis for each guideline will enable development groups.

Annex A provides a template for the information that should be supplied with each guideline.

It is unlikely that all of the guidelines given in this technical report will be applicable to all application domains.

B.2 Documenting of the selection process

The intended purpose of this documentation is to enable third parties to evaluate:

- the effectiveness of the process that created each guideline,
- the applicability of individual guidelines to a particular project.

Annex C
(informative)
Template for use in proposing programming language vulnerabilities

C. Skeleton template for use in proposing programming language vulnerabilities

C.1 6.<*x*> <*unique immutable identifier*> <*short title*>

Notes on template header. The number "x" depends on the order in which the vulnerabilities are listed in Clause 6. It will be assigned by the editor. The "unique immutable identifier" is intended to provide an enduring identifier for the vulnerability description, even if their order is changed in the document. The "short title" should be a noun phrase summarizing the description of the application vulnerability. No additional text should appear here.

C.1.0 6.<x>.0 Status and history

The header will be removed before publication.

This temporary section will hold the edit history for the vulnerability. With the current status of the vulnerability.

C.1.1 6.<*x*>.1 Description of application vulnerability

Replace this with a brief description of the application vulnerability. It should be a short paragraph.

C.1.2 6.<*x*>.2 Cross reference

CWE: *Replace this with the CWE identifier. At a later date, other cross-references may be added.*

C.1.3 6.<*x*>.3 Categorization

See clause 5.?. *Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*

C.1.4 6.<*x*>.4 Mechanism of failure

Replace this with a brief description of the mechanism of failure. This description provides the link between the programming language vulnerability and the application vulnerability. It should be a short paragraph.

C.1.5 6.<*x*>.5 Applicable language characteristics

Replace this with a description of the various points at which the chain of causation could be broken. It should be a short paragraph.

C.1.6 6.<*x*>.6 Assumed variations among languages

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

Replace this with a bullet list summarizing the pertinent range of characteristics of languages for which this discussion is applicable. This list is intended to assist readers attempting to apply the guidance to languages that have not been treated in the language-specific annexes.

C.1.7 6.<*x*>.7 Implications for standardization

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

Replace this with a bullet list summarizing various ways in which programmers can avoid the vulnerability or contain its bad effects. Begin with the more direct, concrete, and effective means and then progress to the more indirect, abstract, and probabilistic means.

C.1.8 6.<*x*>.8 Bibliography

<Insert numbered references for other documents cited in your description. These will eventually be collected into an overall bibliography for the TR. So, please make the references complete. Someone will eventually have to reformat the references into an ISO-required format, so please err on the side of providing too much information rather than too little. Here [1] is an example of a reference:

[1] Greg Hoglund, Gary McGraw, Exploiting Software: How to Break Code, ISBN-0-201-78695-8, Pearson Education, Boston, MA, 2004

Annex D
(informative)
Template for use in proposing application vulnerabilities

D. Skeleton template for use in proposing application vulnerabilities

D.1 7.<*x*> <*unique immutable identifier*> <*short title*>

Notes on template header. The number "x" depends on the order in which the vulnerabilities are listed in Clause 6. It will be assigned by the editor. The "unique immutable identifier" is intended to provide an enduring identifier for the vulnerability description, even if their order is changed in the document. The "short title" should be a noun phrase summarizing the description of the application vulnerability. No additional text should appear here.

D.1.0 7.<x>.0 Status and history

The header will be removed before publication.

This temporary section will hold the edit history for the vulnerability. With the current status of the vulnerability.

D.1.1 7.<*x*>.1 Description of application vulnerability

Replace this with a brief description of the application vulnerability. It should be a short paragraph.

D.1.2 7.<*x*>.2 Cross reference

CWE: *Replace this with the CWE identifier. At a later date, other cross-references may be added.*

D.1.3 7.<*x*>.3 Categorization

See clause 5.?. *Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*

D.1.4 7.<*x*>.4 Mechanism of failure

Replace this with a brief description of the mechanism of failure. This description provides the link between the programming language vulnerability and the application vulnerability. It should be a short paragraph.

D.1.5 7.<*x*>.5 Assumed variations among languages

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

Replace this with a bullet list summarizing the pertinent range of characteristics of languages for which this discussion is applicable. This list is intended to assist readers attempting to apply the guidance to languages that have not been treated in the language-specific annexes.

D.1.6 7.<*x*>.6 Implications for standardization

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

Replace this with a bullet list summarizing various ways in which programmers can avoid the vulnerability or contain its bad effects. Begin with the more direct, concrete, and effective means and then progress to the more indirect, abstract, and probabilistic means.

D.1.7 7.<*x*>.7 Bibliography

<Insert numbered references for other documents cited in your description. These will eventually be collected into an overall bibliography for the TR. So, please make the references complete. Someone will eventually have to reformat the references into an ISO-required format, so please err on the side of providing too much information rather than too little. Here [1] is an example of a reference:

[1] Greg Hoglund, Gary McGraw, Exploiting Software: How to Break Code, ISBN-0-201-78695-8, Pearson Education, Boston, MA, 2004

Annex E
(informative)
Vulnerability Outline

E. Vulnerability Outline

E.1. Human Factors

E.1.1. BRS-PENDING-leveraging-human-experience

E.2. Environment
- E.2.1. XYN-PENDING-privilege-management
- E.2.2. XYO-PENDING-privilege-sandbox-issues
- E.2.3. Interactions with environment

E.3. Core Language Issues
- E.3.1. BQF-PENDING-unspecified-behavior
- E.3.2. EWF-PENDING-undefined-behavior
- E.3.3. FAB-PENDING-implementation-defined-behavior
- E.3.4. MEM-PENDING-deprecated-features
- E.3.5. BVQ-PENDING-unspecified-functionality

E.4. Documentation

E.5. Preprocessor
- E.5.1. NMP-PENDING-preprocessor-directives

E.6. Declarations and Definitions
- E.6.1. NAI-PENDING-choice-of-clear-names
- E.6.2. AJN-PENDING-choice-of-filenames-and-other-external-identifiers
- E.6.3. XYR-PENDING-unused-variable
- E.6.4. YOW-PENDING-identifier-name-reuse

E.7. Types
- E.7.1. Representation
  - E.7.1.1. IHN-PENDING-strong-typing
  - E.7.1.2. STR-PENDING-bit-representations
- E.7.2. Constants
- E.7.3. Floating point
  - E.7.3.1. PLF-PENDING-floating-point-arithmetic
- E.7.4. Enumerated Types
  - E.7.4.1. CCB-PENDING-enumerator-issues
- E.7.5. Integers
  - E.7.5.1. XYE-PENDING-integer-coercion-errors
- E.7.6. Characters and strings
- E.7.7. Arrays
  - E.7.7.1. XYX-PENDING-boundary-beginning-violation
  - E.7.7.2. XYZ-PENDING-unchecked-array-indexing
  - E.7.7.3. XYW-PENDING-buffer-overflow-in-stack
  - E.7.7.4. XZB-PENDING-buffer-overflow-in-heap
- E.7.8. Structures and Unions
- E.7.9. Pointers
  - E.7.9.1. HFC-PENDING-pointer-casting-and-pointer-type-changes
  - E.7.9.2. RVG-PENDING-pointer-arithmetic
  - E.7.9.3. XYH-PENDING-null-pointer-dereference
  - E.7.9.4. XYK-PENDING-dangling-reference-to-heap
- E.7.10. Vectors

E.8. Objects

E.9. Templates/Generics
- E.9.1. SYM-PENDING-templates-and-generics

E.16.2. XZS-PENDING-missing-required-cryptographic-step
E.16.3. Authentication
E.16.3.1. XYM-PENDING-insufficiently-protected-credentials
E.16.3.2. XZN-PENDING-missing-or-inconsistent-access-control
E.16.3.3. XZO-PENDING-authentication-logic-error
E.16.3.4. XYP-PENDING-hard-coded-password

1

Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2001

[2] ISO/IEC TR 10000-1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241, *International terminology standards — Preparation and layout*

[4] ISO/IEC TR 15942:2000, "Information technology - Programming languages - Guide for the use of the Ada programming language in high integrity systems"

[5] Joint Strike Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

[6] ISO/IEC 9899:1999, *Programming Languages* – C

[7] ISO/IEC 1539-1:2004, *Programming Languages* – Fortran

[8] ISOISO/IEC 8652:1995/Cor 1:2001/Amd 1:2007, Information technology -- *Programming languages* – Ada

[9] ISO/IEC 15291:1999, Information technology - Programming languages - Ada Semantic Interface Specification (ASIS)

[10] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

[11] IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

[12] ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

[13] J Barnes. High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[14] R. Seacord Preliminary draft of the CERT C Programming Language Secure Coding Standard. ISO/IEC JTC 1/SC 22/OWGV N0059, April 2007.

[15] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2004 (second edition)[2].

[16] ISO/IEC TR24731-1, *Extensions to the C Library, — Part I: Bounds-checking interfaces*

[17] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[2] The first edition should not be used or quoted in this work.