ISO/IEC JTC 1/SC 22/OWGV N0095

PDTR 24772, 06 August 2007

ISO/IEC JTC 1/SC 22 N 0000

Date: 2007-06-29

ISO/IEC PDTR 24772

ISO/IEC JTC 1/SC 22/OWG

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: (20) Preparation
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

1) Applicable to the computer programming languages covered in this document.
2) Applicable to software written, reviewed and maintained for any application.
3) Applicable in any context where assured behavior is required, e.g. security, safety, mission/business criticality etc.

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 the 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.

Some of the reasons why a guideline might 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. Specific examples of such communities include:

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 includes 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 and who need to use computation as part of their work. These programmers include scientists, engineers, economists, and statisticians. These programmers 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.

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 more efficient execution on a wider range of hardware, or enable use of the language in a wider variety of circumstances.

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.

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 not be specified 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.

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

6.1 XYE Integer Coercion Errors

6.1.0 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

6.1.1 Description of application vulnerability

Integer coercion refers to a set of flaws pertaining to the type casting, extension, or truncation of primitive data types. Common consequences are of integer coercion are undefined states of execution resulting in infinite loops or crashes, or exploitable buffer overflow conditions, resulting in the execution of arbitrary code.

6.1.2 Cross reference

CWE:
192. Integer Coercion Error

6.1.3 Categorization

See clause 5.?.
Group: Arithmetic

6.1.4 Mechanism of failure

Several flaws fall under the category of integer coercion errors. For the most part, these errors in and of themselves result only in availability and data integrity issues. However, in some circumstances, they may result in other, more complicated security related flaws, such as buffer overflow conditions.

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

6.1.5 Range of language characteristics considered

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

- Languages that allow implicit type conversion (coercion).
- Languages that are weakly typed. Strongly typed languages do a strict enforcement of type rules since all types are known at compile time.
- Languages that support logical, arithmetic, or circular shifts. Some languages do not support one or more of the shift types.
- Some languages throw exceptions on ambiguous data casts.

6.1.6 Avoiding the vulnerability or mitigating its effects

[Note: *RSIZE_T* and *verifiably representation* should be considered, see ISO/IEC TR 24731.]

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

- Integer values used in any of the following ways must be guaranteed correct:

    - 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 critical code

- 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.

- 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 throws exceptions on ambiguous data casts 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 in order to reduce the plausibility of error in use.

- Type conversions occur explicitly as the result of a cast or implicitly as required by an operation. While conversions are generally required for the correct execution of a program, they can also lead to lost or misinterpreted data.

- Do not assume that a right shift operation is implemented as either an arithmetic (signed) shift or a logical (unsigned) shift. If `E1` in the expression `E1 >> E2` has a signed type and a negative value, the resulting value is implementation defined and may be either an arithmetic shift or a logical shift. Also, be careful to avoid undefined behavior while performing a bitwise shift.

- Integer conversions, including implicit and explicit (using a cast), must be guaranteed not to result in lost or misinterpreted data. The only integer type conversions that are guaranteed to be safe for all data values and all possible conforming implementations are conversions of an integral value to a wider type of the same signedness. Typically, converting an integer to a smaller type results in truncation of the high-order bits.

- Bitwise shifts include left shift operations of the form *shift-expression* << *additive-expression* and right shift operations of the form *shift-expression* >> *additive-expression*. The integer promotions are performed on the operands, each of which has integer type. The type of the result is that of the promoted left operand. If the value of the right operand is negative or is greater than or equal to the width of the promoted left operand, the behavior is undefined. [Bitwise shifting may be a distinct vulnerability.]

- If an integer expression is compared to, or assigned to, a larger integer size, then that integer expression should be evaluated in that larger size by explicitly casting one of the operands.

6.1.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.1.8 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

6.2 XYF Numeric Truncation Error

[Note: Consider combining with XYE.]

6.2.0 Status and history

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

6.2.1 Description of application vulnerability

Truncation errors occur when a primitive is cast to a primitive of a smaller size and data is lost in the conversion.

6.2.2 Cross reference

CWE:
197. Numeric Truncation Error

6.2.3 Categorization

See clause 5.?.
Group: Arithmetic

6.2.4 Mechanism of failure

When a primitive is cast to a smaller primitive, the high order bits of the large value are lost in the conversion. If high order bits are lost, then the new primitive will have lost some of the value of the original primitive, resulting in a value that could cause unintended consequences. For instance, the new primitive may used as an index into a buffer, a loop iterator, or simply as necessary state data. In any case, the value cannot be trusted and the system will be in an undefined state. While this method may be employed viably to isolate the low bits of a value, this usage is rare and better methods are available for isolating bits such as masking.

6.2.5 Range of language characteristics considered

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

- Languages that allow implicit type conversion (coercion).
- Languages that are weakly typed. Strongly typed languages do a strict enforcement of type rules since all types are known at compile time.
- Languages that do not throw exceptions on ambiguous data casts.

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:

- Ensure that no casts, implicit or explicit, take place that move from a larger size primitive to a smaller size primitive.
- Should the isolation of smaller bits of a value be desired, masking of the original value is safer and more predictable.

6.2.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.2.8 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

6.3 XYG Value Problems

[Note: Consider merging with XZM.]

6.3.0 Status and history

IN
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

6.3.1 Description of application vulnerability

The software does not properly handle the case where the number of parameters, fields or argument names is different from the number provided.

6.3.2 Cross reference

CWE:
230. Missing Value Error
231. Extra Value Error

6.3.3 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.>*

6.3.4 Mechanism of failure

The software does not properly handle the case where the number of parameters, fields or argument names is different from the number provided. In the case of too few, a parameter, field or argument name is specified, but the associated value is empty, blank or null. Alternatively, in the case of too many, more values are specified than expected. This typically occurs in situations when only one value is expected.

6.3.5 Range of language characteristics considered

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

- Languages that do not pass `NULL` as the value of a parameter if too few arguments are provided.
- Languages that do not require the number and type of parameters to be equal to the parameters provided.

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 using input provided, check that the number of parameters, fields or argument names provided is equal to the number expected.

6.3.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.3.8 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

6.4 XYH Null Pointer Dereference

6.4.0 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

6.4.1 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.

6.4.2 Cross reference

CWE:
467. Null Pointer Dereference

6.4.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.4.4 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.

6.4.5 Range of language characteristics considered

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.

6.4.6 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`.

6.4.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.4.8 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

6.5 XYK Pointer Use After Free

6.5.0 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

6.5.1 Description of application vulnerability

Calling `free()` twice on the same memory address can lead to a buffer overflow or referencing memory after it has been freed can cause a program to crash.

6.5.2 Cross reference

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

[Note: perhaps double free and use after free should be separate items.]

6.5.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.5.4 Mechanism of failure

Doubly freeing memory may result in allowing an attacker to execute arbitrary code. The use of previously freed memory may corrupt valid data, if the memory area in question has been allocated and used properly elsewhere. If chunk consolidation occurs after the use of previously freed data, the process may crash when invalid data is used as chunk information. If malicious data is entered before chunk consolidation can take place, it may be possible to take advantage of a write-what-where primitive to execute arbitrary code.

When a program calls `free()` twice with the same argument, the program's memory management data structures become corrupted. This corruption can cause the program to crash or, in some circumstances, cause two later calls to `malloc()` to return the same pointer. If `malloc()` returns the same value twice and the program later gives the attacker control over the data that is written into this doubly-allocated memory, the program becomes vulnerable to a buffer overflow attack.

The use of 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 flaw. The simplest way data corruption may occur involves the system's reuse of the freed memory. Like double free errors and memory leaks, Use After Free errors have 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 to another pointer validly at some point after it has been freed. The original pointer to the freed memory is used again and points to somewhere within the new allocation. As the data is changed, it corrupts the validly used memory. This induces undefined behavior in the process. If the newly allocated data chances to hold a class, 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 to valid shell code, execution of arbitrary code can be achieved.

The lifetime of an object is the portion of program execution during which storage is guaranteed to be reserved for it. An object exists, has a constant address, and retains its last-stored value throughout its lifetime. If an object is referred to outside of its lifetime, the behavior is undefined. The value of a pointer becomes indeterminate when the object it points to reaches the end of its lifetime.

6.5.5 Range of language characteristics considered

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.

6.5.6 Avoiding the vulnerability or mitigating its effects

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

- 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 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 writing to unallocated memory. 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.

6.5.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.5.8 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

6.6 XYL Memory Leak

6.6.0 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

6.6.1 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.

6.6.2 Cross reference

CWE:
401. Memory Leak

6.6.3 Categorization

See clause 5.?.
Group: Dynamic Allocation

6.6.4 Mechanism of failure

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.

6.6.5 Range of language characteristics considered

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

- Languages that can dynamically allocate memory.
- Languages that do not have the capability for garbage collection to collect dynamically allocated memory that is no longer reachable.

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:

- 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 such as Boehm-Demers-Weiser Garbage Collector or Valgrind. 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 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.

- Memory leaks can be eliminated by avoiding the use of dynamically allocated storage entirely.

Note: some consider this to be a design issue rather than a coding issue.

6.6.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.6.8 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

6.7 XYW Buffer Overflow in Stack

[Note: Consider merging this with XZB.]

6.7.0 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

6.7.1 Description of application vulnerability

A buffer overflow in the stack condition occurs when the buffer being overwritten is allocated on the stack (i.e., is a local variable or, rarely, a parameter to a function).

6.7.2 Cross reference

CWE:
121. Stack Overflow

6.7.3 Categorization

See clause 5.?.
Group: Array Bounds

6.7.4 Mechanism of failure

There are generally several security-critical data on an execution stack that can lead to arbitrary code execution. The most prominent is the stored return address, the memory address at which execution should continue once the current function is finished executing. The attacker can overwrite this value with some memory address to which the attacker also has write access, into which he places arbitrary code to be run with the full privileges of the vulnerable program. Alternately, the attacker can supply the address of an important call, for instance the POSIX `system()` call, leaving arguments to the call on the stack. This is often called a return into libc exploit, since the attacker generally forces the program to jump at return time into an interesting routine in the C library (libc). Other important data commonly on the stack include the stack pointer and frame pointer, two values that indicate offsets for computing memory addresses. Modifying those values can often be leveraged into a "write-what-where" condition.

Stack overflows can instantiate in return address overwrites, stack pointer overwrites or frame pointer overwrites. They can also be considered function pointer overwrites, array indexer overwrites or write-what-where condition, etc.

Buffer overflows can be exploited for a variety of purposes. A relatively easy way of exploitation is to overflow a buffer so it leads to a crash. Other attacks leading to lack of availability are possible, including putting the program into an infinite loop. Buffer overflows often can be used to execute arbitrary code. When the consequence is arbitrary code execution, this can often be used to subvert any other security service.

6.7.5 Range of language characteristics considered

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

- Some languages or compilers perform or implement automatic bounds checking.
- 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:

- Although not a complete solution, an abstraction library to abstract away risky APIs can be used.
- Compiler-based canary mechanisms such as StackGuard, ProPolice and the Microsoft Visual Studio /GS flag can be used. However, unless automatic bounds checking is provided, it is not a complete solution.

- OS-level preventative functionality can also be used.

6.7.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.7.8 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

6.8 XZB Buffer Overflow in Heap

6.8.0 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

6.8.1 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.

6.8.2 Cross reference

CWE:
122. Heap Overflow

6.8.3 Categorization

See clause 5.?.
Group: Array Bounds

6.8.4 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.

6.8.5 Range of language characteristics considered

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.)

6.8.6 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.

6.8.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.8.8 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

6.9 XZM Missing Parameter Error [Could also be Parameter Signature Mismatch]

6.9.0 Status and history

IN
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

6.9.1 Description of application vulnerability

If too few arguments are sent to a function, the function will still pop the expected number of arguments from the stack. A variable number of arguments could potentially be exhausted by a function.

6.9.2 Cross reference

CWE:
234. Missing Parameter Error

6.9.3 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.>*

6.9.4 Mechanism of failure

There is the potential for arbitrary code execution with privileges of the vulnerable program if function parameter list is exhausted or the program could potentially fail if it needs more arguments then are available.

[Note: Linking separately compiled modules can be a problem. Using an object code library can be a problem.]

6.9.5 Range of language characteristics considered

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

- Languages that do not pass `NULL` as the value of a parameter if too few arguments are provided.
- Languages that do not require the number and type of parameters to be equal to the parameters provided.

6.9.6 Avoiding the vulnerability or mitigating its effects

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

- Forward declare all functions. Forward declaration of all used functions will result in a compiler error if too few arguments are sent to a function.
- Some languages have facilities to assist in linking to other languages or to separately compiled modules.

6.9.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.9.8 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

6.10 XYY Wrap-around Error

6.10.0 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

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.

6.10.2 Cross reference

CWE:
128. Wrap-around Error

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 storage space, the system will fail to recognize this [not categorically correct], 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.

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 Range of language characteristics considered

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

- Some languages trigger an exception condition when a wrap-around error occurs.

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.

6.10.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.10.8 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

6.11 XYQ Expression Issues

6.11.0 Status and history

IN
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

The software contains an expression that will always evaluate to the same Boolean value (either always true or always false).

[Note: This might be generalized to a discussion of "redundant" code and/or "dead" code. Some prefer this be phrased in terms of "unreachable code".]

[From DO-178B:

Dead code – Executable object code (or data) which, as a result of a design error 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. An exception is embedded identifiers.

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.]

6.11.2 Cross reference

CWE:

570. Expression is Always True
571. Expression is Always False

6.11.3 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.>*

6.11.4 Mechanism of failure

Any boolean expression that evaluates to the same value is indicative of superfluous code and is possibly indicative of a bug that exists and, although the chance is remote, possibly could be exploited.

6.11.5 Range of language characteristics considered

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

- All languages that have Boolean expressions are susceptible to this.

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:

- This expression will always evaluate to the same Boolean value meaning the program could be rewritten in a simpler form. The nearby code may be present for debugging purposes, or it may not have been maintained along with the rest of the program. Coding guidelines could require the programmer to declare whether such instances are intentional.

- The expression could be indicative of an earlier bug earlier and additional testing may be needed to ascertain why the same Boolean value is occurring.

[Note: This relates to the DO-178B distinction between "dead" code and "deactivated" code. See minutes of Meeting #5 for definitions.]

6.11.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.11.8 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

6.12 XYR Unused Variable

6.12.0 Status and history

IN
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

The variable's value is assigned but never used or never assigned at all, making it a dead store.

6.12.2 Cross reference

CWE:
563. Unused Variable

6.12.3 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.>*

6.12.4 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. Note that this may be acceptable if it is a volatile variable. An unused variable is unlikely to be the cause of a vulnerability, however it is indicative of a lack of a clean compile at a reasonably high level of compiler settings.

6.12.5 Range of language characteristics considered

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

- Only static typed programming languages are susceptible to declaring a variable but never using it. Closely related is directly assigning a value to a variable in a dynamic typed programming language and never referencing the variable again.

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:

- Most compilers can detect unused variables. However, the detection may have to be enabled as the default may be to ignore unused variables.

6.12.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.12.8 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

6.13 XYX Boundary Beginning Violation

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

6.13.0 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

6.13.1 Description of application vulnerability

A buffer underwrite condition occurs when a buffer is indexed with a negative number, or pointer arithmetic with a negative value results in a position before the beginning of the valid memory location.

6.13.2 Cross reference

CWE:
124. Boundary Beginning Violation ("buffer underwrite")

6.13.3 Categorization

See clause 5.?.
Group: Array Bounds

6.13.4 Mechanism of failure

Buffer underwrites will very likely result in the corruption of relevant memory, and perhaps instructions, leading to a crash. If the memory corrupted memory can be effectively controlled, it may be possible to execute arbitrary code. If the memory corrupted is data rather than instructions, the system will continue to function with improper changes, ones made in violation of a policy, whether explicit or implicit.

6.13.5 Range of language characteristics considered

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.)

6.13.6 Avoiding the vulnerability or mitigating its effects

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

- Some languages have facilities or add-on options that can be used to automatically check array indexes.

- Add-on tools, such as static analyzers, can be used to detect possible violations. Coding techniques can be used and encouraged through their specification in coding guidelines that improve the analyzability of the code.

- Sanity checks should be performed on all calculated values used as index or for pointer arithmetic.

6.13.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.13.8 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

6.14 XZI Sign Extension Error

6.14.0 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

6.14.1 Description of application vulnerability

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

[Note: combining XYE, XYF, XYY, XZI as "integer arithmetic" was suggested.]

[Note: Should "divide by zero" be added?]

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 such 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.

6.14.5 Range of language characteristics considered

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 or standard function to extend signed numbers.

- When extending signed numbers fill in the new bits with 0 if the sign bit is 0 or fill the new bits with 1 if the sign bit is 1.

- Cast a character as unsigned before conversion to an integer.

6.14.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.14.8 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

6.15 XZH Off-by-one Error

6.15.0 Status and history

IN
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.

[Note: This may need further study. For example, this might be an umbrella for a lot of individual items. On the other hand, this might be a contributing cause of other items.]

6.15.2 Cross reference

CWE:
193. Off-by-one Error

6.15.3 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.>*

6.15.4 Mechanism of failure

This could lead to a buffer overflow. However that is not always the case. For example, an off-by-one error could be a factor in a partial comparison, a read from the wrong memory location, or an incorrect conditional.

6.15.5 Range of language characteristics considered

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

- Many languages have mechanisms to assist in the problem, e.g. methods to obtain the actual bounds of an array.

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 very 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.

6.15.7 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.15.8 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

6.16 XYZ Unchecked Array Indexing

[Note: Perhaps XYW, XYX, XYZ and XZB should be combined into two items: array indexing violations when accessing individual elements and block move/copy.]

6.16.0 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

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 Range of language characteristics considered

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

6.16.8 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

7. Application Vulnerabilities

7.1 XYU Using Hibernate to Execute SQL

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 attackers 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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.1.7 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

7.2 XYA Relative Path Traversal

7.2.0 History and status

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

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.2.7 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

7.3 XYP Hard-coded Password

7.3.0 History and status

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.3.7 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

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.4.7 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

7.5 XYM Insufficiently Protected Credentials

7.5.0 History and status

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.5.7 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

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.6.7 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

7.7 XYN Privilege Management

7.7.0 History and status

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.7.7 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

7.8 XYO Privilege Sandbox Issues

7.8.0 History and status

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.8.7 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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.9.7 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

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

<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

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

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 a resource exhaustion occurs.

7.11.6 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.11.7 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

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.12.7 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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.13.7 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

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

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

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.14.7 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

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

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.]

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.15.7 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

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

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:

<Replace this with a bullet list summarizing various ways in which programmers can avoid the programming language vulnerability, break the chain of causation to the application vulnerability, or contain the bad effects of the application vulnerability. Begin with the more direct, concrete, and effective means and then progress to the more indirect, abstract, and probabilistic means.>

7.16.6 Implications for standardization

<Recommendations for other working groups will be recorded here. For example, we might record suggestions for changes to language standards or API standards.>

7.16.7 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 A
(informative)

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

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 Range of language characteristics considered

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.1.7 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.1.8 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

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)[1].

[16] ISO/IEC TR24732, *Extensions to the C Library, — Part I: Bounds-checking interfaces*

[17] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[1] The first edition should not be used or quoted in this work.

