# ISO/IEC JTC 1/SC 22/WG 23 N 0408

***Proposal to merge XZI and FLC***

| | |
|---|---|
| **Date** | 22 June 2012 |
| **Contributed by** | Benito, Keaton, Plum |
| **Original file name** | ca22-trying-again.docx |
| **Notes** | Possible resolution of CA-22 in PDTR ballot of edition 2 |

**[This is the existing rule XZI … it refers to CWE 194 …**
**http://cwe.mitre.org/data/definitions/194.html , so start by looking at that one**
**]**

## 6.18 Sign Extension Error [XZI]

### 6.18.1 Description of application vulnerability

Extending a signed variable that holds a negative value may produce an incorrect result.

### 6.18.2 Cross reference

CWE:
194. Incorrect Sign Extension
MISRA C++ 2008: 5-0-4
CERT C guidelines: INT13-C

### 6.18.3 Mechanism of failure

Converting a signed data type to a larger data type or pointer can cause unexpected behaviour 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 type short or a signed integer (32-bit) is converted to an integer type long (64-bit). 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.

### 6.18.4 Applicable language characteristics

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

☐ Languages that are weakly typed due to their lack of enforcement of type classifications and interactions.

☐ Languages that explicitly or implicitly allow applying unsigned extension operations to signed entities or vice versa.

### 6.18.5 Avoiding the vulnerability or mitigating its effects

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

☐ Use a sign extension library, standard function, or appropriate language-specific coding methods to extend signed values.

☐ Use static analysis tools to help locate situations in which the conversion of variables might have unintended consequences.

### 6.18.6 Implications for standardization

In future standardization activities, the following items should be considered:

☐ Language definitions should define implicit and explicit conversions in a way that prevents alteration of the mathematical value beyond traditional rounding rules.

## [This is the existing FLC …]

## 6.7 Numeric Conversion Errors [FLC]

### 6.7.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types [32]:

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

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

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

Type conversion errors can lead to erroneous data being generated, algorithms that fail to terminate, array bounds errors, and arbitrary program execution.

### 6.7.2 Cross reference

CWE:

192. Integer Coercion Error

MISRA C 2004: 10.1-10.6, 11.3-11.5, and 12.9

MISRA C++ 2008: 2-13-3, 5-0-3, 5-0-4, 5-0-5, 5-0-6, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-2-5, 5-2-9, and 5-3-2

CERT C guidelines: FLP34-C, INT02-C, INT08-C, INT31-C, and INT35-C

### 6.7.3 Mechanism of failure

Numeric conversion errors results in data integrity issues, but they may also result in a number of safety and security vulnerabilities. Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. These can result in safety issues, for example, when the Ariane 5 launcher failure occurred due to an improperly handled conversion error resulting in the processor being shutdown [29].

Conversion errors can also result in security issues. An attacker may input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value may then be used as an array index, a loop iterator, a length, a size, state data, or in some other security critical manner. For example, a truncated integer value may be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow [30].

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

### 6.7.4 Applicable language characteristics

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

☐ Languages that perform implicit type conversion (coercion).

☐ Weakly typed languages that do not strictly enforce type rules.

☐ Languages that support logical, arithmetic, or circular shifts on integer values.
☐ Languages that do not generate exceptions on problematic conversions.

### 6.7.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:
☐ The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. All integer values originating from a source that is not trusted should be validated for correctness. 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 [30].

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

☐ The use of static analysis can often identify whether or not unacceptable numeric conversions will occur.

Verifiably in-range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. Faced with a numeric conversion error, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce a numeric value that is within the range of representable values on that system. The latter semantics may be preferable in some situations in that it allows the computation to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what numeric result to return to the user.
A recent innovation from ISO/IEC TR 24731-1 [13] is the definition of the `rsize_t` type for the C programming language. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.

### 6.7.6 Implications for standardization

In future standardization activities, the following items should be considered:
☐ Languages should consider providing means similar to the ISO/IEC TR 24731-1 definition of `rsize_t` type for C to restrict object sizes so as to expose programming errors.

☐ Languages should consider making all type conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.

## [Here is a revision of FLC that covers CWE 194 as well as the existing 192…]

## 6.7 Numeric Conversion Errors [FLC]

### 6.7.1 Description of application vulnerability

Certain contexts in various languages may require exact matches with respect to types [32]:

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

Type conversion seeks to follow these exact match rules while allowing programmers some flexibility in using values such as: structurally-equivalent types in a name-equivalent language, types whose value ranges may be distinct but intersect (for example, subranges), and distinct types with sensible/meaningful corresponding values (for example,

integers and floats). Explicit conversions are called *type casts*. An implicit type conversion between compatible but not necessarily equivalent types is called *type coercion*.
Numeric conversions can lead to a loss of data, if the target representation is not capable of representing the original value. For example, converting from an integer type to a smaller integer type can result in truncation if the original value cannot be represented in the smaller size and converting a floating point to an integer can result in a loss of precision or an out-of-range value.

[NEW STUFF ...] Converting a signed data type to an unsigned data type or pointer, or an unsigned data type to a signed data type, can cause unexpected behaviour. These conversion 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.

[BACK TO EXISTING ...] Type conversion errors can lead to erroneous data being generated, algorithms that fail to terminate, array bounds errors, and arbitrary program execution.

## 6.7.2 Cross reference

CWE:
192. Integer Coercion Error
[ALSO ADD ...] 194. Incorrect Sign Extension
[ALSO ADD ...] MISRA C++ 2008: 5-0-4 [AND] CERT C guidelines: INT13-C

MISRA C 2004: 10.1-10.6, 11.3-11.5, and 12.9
MISRA C++ 2008: 2-13-3, 5-0-3, 5-0-4, 5-0-5, 5-0-6, 5-0-7, 5-0-8, 5-0-9, 5-0-10, 5-2-5, 5-2-9, and 5-3-2
CERT C guidelines: FLP34-C, INT02-C, INT08-C, INT31-C, and INT35-C

## 6.7.3 Mechanism of failure

Numeric conversion errors results in data integrity issues, but they may also result in a number of safety and security vulnerabilities. Vulnerabilities typically occur when appropriate range checking is not performed, and unanticipated values are encountered. These can result in safety issues, for example, when the Ariane 5 launcher failure occurred due to an improperly handled conversion error resulting in the processor being shutdown [29].
Conversion errors can also result in security issues. An attacker may input a particular numeric value to exploit a flaw in the program logic. The resulting erroneous value may then be used as an array index, a loop iterator, a length, a size, state data, or in some other security critical manner. For example, a truncated integer value may be used to allocate memory, while the actual length is used to copy information to the newly allocated memory, resulting in a buffer overflow [30].
Numeric type conversion errors often lead to undefined states of execution resulting in infinite loops or crashes. In some cases, integer type conversion errors can lead to exploitable buffer overflow conditions, resulting in the execution of arbitrary code. Integer type conversion errors result in an incorrect value being stored for the variable in question.

## 6.7.4 Applicable language characteristics

This vulnerability description is intended to be applicable to languages with the following characteristics:
☐ Languages that perform implicit type conversion (coercion).
☐ Weakly typed languages that do not strictly enforce type rules.
☐ Languages that support logical, arithmetic, or circular shifts on integer values.
☐ Languages that do not generate exceptions on problematic conversions.

## 6.7.5 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:
☐ The first line of defense against integer vulnerabilities should be range checking, either explicitly or through strong typing. All integer values originating from a source that is not trusted should be validated for correctness. 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 [30].

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

☐ The use of static analysis can often identify whether or not unacceptable numeric conversions will occur.

Verifiably in-range operations are often preferable to treating out of range values as an error condition because the handling of these errors has been repeatedly shown to cause denial-of-service problems in actual applications. Faced with a numeric conversion error, the underlying computer system may do one of two things: (a) signal some sort of error condition, or (b) produce a numeric value that is within the range of representable values on that system. The latter semantics may be preferable in some situations in that it allows the computation to proceed, thus avoiding a denial-of-service attack. However, it raises the question of what numeric result to return to the user.
A recent innovation from ISO/IEC TR 24731-1 [13] is the definition of the `rsize_t` type for the C programming language. Extremely large object sizes are frequently a sign that an object's size was calculated incorrectly. For example, negative numbers appear as very large positive numbers when converted to an unsigned type like `size_t`. Also, some implementations do not support objects as large as the maximum value that can be represented by type `size_t`. For these reasons, it is sometimes beneficial to restrict the range of object sizes to detect programming errors. For implementations targeting machines with large address spaces, it is recommended that `RSIZE_MAX` be defined as the smaller of the size of the largest object supported or `(SIZE_MAX >> 1)`, even if this limit is smaller than the size of some legitimate, but very large, objects. Implementations targeting machines with small address spaces may wish to define `RSIZE_MAX` as `SIZE_MAX`, which means that there is no object size that is considered a runtime-constraint violation.

### 6.7.6 Implications for standardization

In future standardization activities, the following items should be considered:

☐ Languages should consider providing means similar to the ISO/IEC TR 24731-1 definition of `rsize_t` type for C to restrict object sizes so as to expose programming errors.

☐ Languages should consider making all type conversions explicit or at least generating warnings for implicit conversions where loss of data might occur.