INT35-C. Use correct integer precisions

Integer types in C have both a *size* and a *precision*. The size indicates the number of bytes used by an object and can be retrieved for any object or type using the \texttt{sizeof} operator. The precision of an integer type is the number of bits it uses to represent values, excluding any sign and padding bits.

Padding bits contribute to the integer's size, but not to its precision. Consequently, inferring the precision of an integer type from its size may result in too large a value, which can then lead to incorrect assumptions about the numeric range of these types. Programmers should use correct integer precisions in their code, and in particular, should not use the \texttt{sizeof} operator to compute the precision of an integer type on architectures that use padding bits or in strictly conforming (that is, portable) programs.

Noncompliant Code Example

This noncompliant code example illustrates a function that produces 2 raised to the power of the function argument. To prevent undefined behavior in compliance with INT34-C. Do not shift an expression by a negative number of bits or by greater than or equal to the number of bits that exist in the operand, the function ensures that the argument is less than the number of bits used to store a value of type \texttt{unsigned int}.

```c
#include <limits.h>

unsigned int pow2(unsigned int exp) {
    if (exp >= sizeof(unsigned int) * CHAR_BIT) {
        /* Handle error */
    }
    return 1 << exp;
}
```

However, if this code runs on a platform where \texttt{unsigned int} has one or more padding bits, it can still result in values for \texttt{exp} that are too large. For example, on a platform that stores \texttt{unsigned int} in 64 bits, but uses only 48 bits to represent the value, a left shift of 56 bits would result in undefined behavior.

Compliant Solution

This compliant solution uses a \texttt{popcount()} function, which counts the number of bits set on any unsigned integer, allowing this code to determine the precision of any integer type, signed or unsigned.

```c
#include <stdint.h>
#include <stddef.h>

/* Returns the number of set bits */
size_t popcount(uintmax_t num) {
    size_t precision = 0;
    while (num != 0) {
        if (num % 2 == 1) {
            precision++;
        }
        num >>= 1;
    }
    return precision;
}
#define PRECISION(umax_value) popcount(umax_value)
```

Implementations can replace the \texttt{PRECISION()} macro with a type-generic macro that returns an integer constant expression that is the precision of the specified type for that implementation. This return value can then be used anywhere an integer constant expression can be used, such as in a static assertion. (See DCL03-C. Use a static assertion to test the value of a constant expression.) The following type generic macro, for example, might be used for a specific implementation targeting the IA-32 architecture:
#define PRECISION(value)  _Generic(value, 
    unsigned char : 8, 
    unsigned short: 16, 
    unsigned int : 32, 
    unsigned long : 32, 
    unsigned long long : 64, 
    signed char : 7, 
    signed short : 15, 
    signed int : 31, 
    signed long : 31, 
    signed long long : 63)

The revised version of the pow2() function uses the PRECISION() macro to determine the precision of the unsigned type:

#include <stddef.h>
#include <stdint.h>
#include <limits.h>
extern size_t popcount(uintmax_t);
#define PRECISION(umax_value) popcount(umax_value)

unsigned int pow2(unsigned int exp) {
    if (exp >= PRECISION(UINT_MAX)) {
        /* Handle error */
    }
    return 1 << exp;
}

Implementation Details

Some platforms, such as the Cray Linux Environment (CLE; supported on Cray XT CNL compute nodes), provide a _popcnt instruction that can substitute for the popcount() function.

#define PRECISION(umax_value) _popcnt(umax_value)

Risk Assessment

Mistaking an integer's size for its precision can permit invalid precision arguments to operations such as bitwise shifts, resulting in undefined behavior.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT35-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>Medium</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>19.04</td>
<td></td>
<td>Supported: Astrée reports overflows due to insufficient precision.</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-INT35-a</td>
<td>Use correct integer precisions when checking the right hand operand of the shift operator</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule INT35-C</td>
<td>Checks for situations when integer precisions are exceeded (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0582</td>
<td></td>
</tr>
</tbody>
</table>

Related Guidelines

Key here (explains table format and definitions)
CERT-CWE Mapping Notes

Key here for mapping notes

**CWE-190 and INT35-C**

Intersection( INT35-C, CWE-190) = Ø

INT35-C used to map to CWE-190 but has been replaced with a new rule that has no overlap with CWE-190.

**CWE-681 and INT35-C**

Intersection(INT35-C, CWE-681) = due to incorrect use of integer precision, conversion from one data type to another causing data to be omitted or translated in a way that produces unexpected values

CWE-681 - INT35-C = list2, where list2 =

- conversion from one data type to another causing data to be omitted or translated in a way that produces unexpected values, not involving incorrect use of integer precision

INT35-C - CWE-681 = list1, where list1 =

- incorrect use of integer precision not related to conversion from one data type to another

**Bibliography**

[Dowd 2006] Chapter 6, "C Language Issues"

[C99 Rationale 2003] 6.5.7, "Bitwise Shift Operators"