INT17-C. Define integer constants in an implementation-independent manner

Integer constants are often used as masks or specific bit values. Frequently, these constants are expressed in hexadecimal form to indicate to the programmer how the data might be represented in the machine. However, hexadecimal integer constants are frequently used in a nonportable manner.

Noncompliant Code Example

In this pedagogical noncompliant code example, the flipbits() function complements the value stored in `x` by performing a bitwise exclusive OR against a mask with all bits set to 1. For implementations where `unsigned long` is represented by a 32-bit value, each bit of `x` is correctly complemented.

```c
/* (Incorrect) Set all bits in mask to 1 */
const unsigned long mask = 0xFFFFFFFF;
unsigned long flipbits(unsigned long x) {
    return x ^ mask;
}
```

However, on implementations where values of type `unsigned long` are represented by greater than 32 bits, `mask` will have leading 0s. For example, on implementations where values of type `unsigned long` are 64 bits long, `mask` is assigned the value `0x00000000FFFFFFFF`. Consequently, only the lower-order bits of `x` are complemented.

Compliant Solution (1)

In this compliant solution, the integer constant `-1` is used to set all bits in `mask` to 1. The integer constant `-1` is of type `signed int`. Because `-1` cannot be represented by a variable of type `unsigned long`, it is converted to a representable number according to the rule in subclause 6.3.1.3, paragraph 2, of the C Standard [ISO/IEC 9899:2011]:

"One more than the maximum value that can be represented in the new type," `ULONG_MAX + 1`, is added to `-1`, resulting in a right-side value of `ULONG_MAX`. The representation of `ULONG_MAX` is guaranteed to have all bits set to 1 by subclause 6.2.6.2, paragraph 1:

```
/* (Correct) Set all bits in mask to 1 */
const unsigned long mask = -1;
unsigned long flipbits(unsigned long x) {
    return x ^ mask;
}
```

Noncompliant Code Example

In this noncompliant code example, a programmer attempts to set the most significant bit:
const unsigned long mask = 0x80000000;
unsigned long x;
/* Initialize x */
x |= mask;

This code has the desired effect for implementations where unsigned long has a precision of 32 bits but not for implementations where unsigned long has a precision of 64 bits.

Compliant Solution

A portable (and safer) way of setting the high-order bit is to use a shift expression, as in this compliant solution:

```c
const unsigned long mask = ~(ULONG_MAX >> 1);
unsigned long x;
/* Initialize x */
x |= mask;
```

Risk Assessment

Vulnerabilities are frequently introduced while porting code. A buffer overflow vulnerability may result, for example, if an incorrectly defined integer constant is used to determine the size of a buffer. It is always best to write portable code, especially when there is no performance overhead for doing so.

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT17-C</td>
<td>High</td>
<td>Probable</td>
<td>Low</td>
<td>P18</td>
<td>L1</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>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-INT17</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

- SEI CERT C++ Coding Standard: VOID INT17-CPP. Define integer constants in an implementation-independent manner

Bibliography

- [Dewhurst 2002]: Gotcha #25, "#define Literals"
- [ISO/IEC 9899:2011]: Subclause 6.2.6, "Representations of Types"
  Subclause 6.3.1.3, "Signed and Unsigned Integers"