MSC15-C. Do not depend on undefined behavior

The C Standard, subclause 3.4.3 [ISO/IEC 9899:2011], defines *undefined behavior* as:

*behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which this International Standard imposes no requirements.*

Subclause 4 explains how the standard identifies undefined behavior (see also *undefined behavior* 1 of Annex J).

Annex J, subclause J.2, "Undefined behavior," enumerates the circumstances under which the behavior of a program is undefined. This list is duplicated on the CC. Undefined Behavior page.

Behavior can be classified as undefined by the C standards committee for the following reasons:

- To give the implementor license not to catch certain program errors that are difficult to diagnose
- To avoid defining obscure corner cases that would favor one implementation strategy over another
- To identify areas of possible conforming language extension: the implementor may augment the language by providing a definition of the officially undefined behavior

Conforming implementations can deal with undefined behavior in a variety of fashions, such as ignoring the situation completely, with unpredictable results; translating or executing the program in a documented manner characteristic of the environment (with or without the issuance of a diagnostic message); or terminating a translation or execution (with the issuance of a diagnostic message). Because compilers are not obligated to generate code for undefined behavior, these behaviors are candidates for optimization. By assuming that undefined behaviors will not occur, compilers can generate code with better performance characteristics.

Increasingly, compiler writers are taking advantage of undefined behaviors in the C programming languages to improve optimizations. These optimizations frequently interfere with the ability of developers to perform cause-effect analysis on their source code—that is, to analyze the dependence of downstream results on prior results. Consequently, these optimizations are eliminating causality in software and are increasing the probability of software faults, defects, and vulnerabilities.

All of this puts the onus on the programmer to develop code that is free from undefined behaviors, with or without the help of the compiler.

Noncompliant Code Example

An example of *undefined behavior* in C is the behavior on signed integer overflow (see also INT32-C. Ensure that operations on signed integers do not result in overflow). This noncompliant code example depends on this behavior to catch the overflow:

```c
#include <assert.h>
#include <limits.h>
#include <stdio.h>

int foo(int a) {
    assert(a + 100 > a);
    printf("%d %d\n", a + 100, a);
    return a;
}

int main(void) {
    foo(100);
    foo(INT_MAX);
    return 0;
}
```

This code checks for signed integer overflow by testing whether \( a + 100 > a \). This test cannot evaluate to false unless an integer overflow occurs. However, because a conforming implementation is not required to generate code for undefined behavior, and signed integer overflow is undefined behavior, this code may be compiled out. For example, GCC 4.1.1 optimizes out the assertion for all optimization levels, and GCC 4.2.3 optimizes out the assertion for programs compiled with `-O2`-level optimization and higher.

On some platforms, the integer overflow causes the program to terminate (before it has an opportunity to test).

Compliant Solution

This compliant solution does not depend on undefined behavior:
#include <assert.h>
#include <limits.h>
#include <stdio.h>

int foo(int a) {
    assert(a < (INT_MAX - 100));
    printf("%d %d\n", a + 100, a);
    return a;
}

int main(void) {
    foo(100);
    foo(INT_MAX);
    return 0;
}

Risk Assessment

Although it is rare that the entire application can be strictly conforming, the goal should be that almost all the code is allowed for a strictly conforming program (which among other things means that it avoids undefined behavior), with the implementation-dependent parts confined to modules that the programmer knows are needed to adapt to the platform when it changes.

<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>MSC15-C</td>
<td>High</td>
<td>Likely</td>
<td>Medium</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>Astrée</td>
<td>19.04</td>
<td></td>
<td>Support: Astrée reports undefined behavior</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>CERT_C-MSC15-a</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Paraso ft C/C++test</td>
<td>10.4.2</td>
<td></td>
<td>Evaluation of constant unsigned integer expressions should not lead to wrap-around</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rec. MSC15-C</td>
<td>Checks for undefined behavior (rec. partially covered)</td>
</tr>
<tr>
<td>PROQA-GA-C</td>
<td>9.7</td>
<td></td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>6.23</td>
<td>V772</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 MSC15-CPP. Do not depend on undefined behavior
| ISO/IEC TR 24772 | Unspecified Behaviour [BQF]  
|                 | Undefined Behaviour [EWF]  
|                 | Implementation-Defined Behaviour [FAB] |

**Bibliography**

|                     | Subclause 4, "Conformance"  
|                     | Subclause J.2, "Undefined Behavior" |
| [Seacord 2013]     | Chapter 5, "Integer Security" |