**MSC14-C. Do not introduce unnecessary platform dependencies**

Platform dependencies may be introduced to improve performance on a particular platform. This can be a dangerous practice, particularly if these dependencies are not appropriately documented during development and addressed during porting. Platform dependencies that have no performance or other benefits should consequently be avoided because they may introduce errors during porting.

The C Standard identifies four different kinds of nonportable behavior. Each section of Annex J of the C Standard enumerates distinct instances of behaviors of each kind.

<table>
<thead>
<tr>
<th>Nonportable Behavior</th>
<th>Definition</th>
<th>Annex J Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unspecified behavior</td>
<td>Behavior for which the standard provides two or more possibilities and imposes no further requirements on which is chosen in any instance.</td>
<td>J.1</td>
</tr>
<tr>
<td>Undefined behavior</td>
<td>Behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which the standard imposes no requirements. An example of undefined behavior is the behavior on signed integer overflow.</td>
<td>J.2</td>
</tr>
<tr>
<td>Implementation-defined behavior</td>
<td>Unspecified behavior whereby each implementation documents how the choice is made.</td>
<td>J.3</td>
</tr>
<tr>
<td>Locale-specific behavior</td>
<td>Behavior that depends on local conventions of nationality, culture, and language that each implementation documents.</td>
<td>J.4</td>
</tr>
</tbody>
</table>

An example of undefined behavior is a null `char*` pointer as an argument to the `printf` function corresponding to the `%s` format specification. Although some implementations (such as the GNU C Library) provide well-defined semantics for this case, others do not, causing programs that rely on this behavior to fail abnormally.

An example of unspecified behavior is the order in which the arguments to a function are evaluated.

An example of implementation-defined behavior is the propagation of the high-order bit when a signed integer is shifted right.

Most legitimate platform dependencies can and should be isolated in separate modules that expose portable, platform-agnostic interfaces to platform-specific implementations. Portable applications that cannot avoid relying on platform-specific details should always provide a generic, portable, standards-based solution as a fallback mechanism for the platform-specific alternative. That way, such an application can be more easily ported to new platforms without an excessive risk of security flaws caused by assumptions that do not hold in the new environment.

**Noncompliant Code Example**

This noncompliant code example uses the complement operator in the test for unsigned integer overflow:

```c
unsigned int ui;
unsigned int ui2;
unsigned int sum;

if (~ui < ui2) {
    /* Handle error condition */
}
sum = ui + ui2;
```

This code assumes that the implementation uses two's complement representation. This assumption is commonly true but is not guaranteed by the standard.

This code sample also violates **INT14-C. Avoid performing bitwise and arithmetic operations on the same data.**

**Compliant Solution**

This compliant solution implements a strictly conforming test for unsigned overflow:

```c
unsigned int ui;
unsigned int ui2;
unsigned int sum;

if (~ui < ui2) {
    /* Handle error condition */
}
sum = ui + ui2;
```
unsigned int ui;
unsigned int ui2;
unsigned int sum;

if (UINT_MAX - ui < ui2) {
    /* Handle error condition */
}
sum = ui + ui2;

If the noncompliant form of this test is truly faster, talk to your compiler vendor because, if these tests are equivalent, optimization should occur. If both forms have the same performance, prefer the portable form.

Noncompliant Code Example (strerror_r)

The GNU libc implementation of `strerror_r` declares the function to return `char*`, in conflict with the POSIX specification. This noncompliant code example relies on this return type to pass the return value as an argument to the `%s` formatting directive to `fprintf`. The behavior of the example will be `undefined` on a platform that declares the return type of `strerror_r()` to be `int`, in accordance with POSIX.

```c
void f() {
    char buf[BUFSIZ];
    fprintf(stderr, "Error: %s\n",
            strerror_r(errno, buf, sizeof buf));
}
```

Compliant Solution (strerror_r)

The compliant solution disables the nonconforming declaration of `strerror_r()` by explicitly requesting POSIX conformance before including the `<string.h>` header that declares the function and handles the function’s failure by copying the "Unknown error" string into the buffer. An alternate solution is to use the `strerror_s()` function defined by Annex K.

Note that the function assigns the result of the call to `strerror_r()` to a variable of type `int`. This assignment is a defense-in-depth strategy guarding against inadvertently invoking `strerror_r()` that returns `char*`; a conforming compiler is required to issue a diagnostic for the ill-formed conversion from `char*` to `int`.

```c
#define _XOPEN_SOURCE 600
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
void f() {
    char buf[BUFSIZ];
    int result;
    result = strerror_r(errno, buf, sizeof buf);
    if (0 != result) {
        strcpy(buf, "Unknown error");
        fprintf(stderr, "Error: %s\n", buf);
    }
}
```

Risk Assessment

Unnecessary platform dependencies are, by definition, unnecessary. Avoiding these dependencies can eliminate porting errors resulting from invalidated assumptions.

<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>MSC14-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>LDRA tool suite</td>
<td>9.7.1</td>
<td>17 D, 69 S, 42 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-MSC14-a</td>
<td>Evaluation of constant unsigned integer expressions should not lead to wrap-around</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0202, 0240, 0241, 0242, 0246, 0284, 0551, 0581, 0601, 0633, 0634, 0635, 0660, 0662, 0830, 0831, 0840, 0899, 1001, 1002, 1003, 1006, 1008, 1012, 1014, 1015, 1019, 1020, 1021, 1022, 1026, 1028, 1029, 1034, 1035, 1036, 1037, 1038, 1041, 1042, 1043, 1044, 1045, 1046, 1434, 3664</td>
<td>Partially implemented</td>
</tr>
</tbody>
</table>

**Related Vulnerabilities**

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

**Related Guidelines**

<table>
<thead>
<tr>
<th>Source</th>
<th>Rule Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CERT C++ Coding Standard</td>
<td>VOID MSC14-CPP. Do not introduce unnecessary platform dependencies</td>
</tr>
<tr>
<td>ISO/IEC TR 24772</td>
<td>Unspecified Behaviour [BQF]</td>
</tr>
</tbody>
</table>

**Bibliography**


[Seacord 2013] Chapter 5, "Integer Security"