FLP02-C. Avoid using floating-point numbers when precise computation is needed

Computers can represent only a finite number of digits. It is therefore impossible to precisely represent repeating binary-representation values such as 1/3 or 1/5 with the most common floating-point representation: binary floating point.

When precise computation is necessary, use alternative representations that can accurately represent the values. For example, if you are performing arithmetic on decimal values and need an exact decimal rounding, represent the values in binary-coded decimal instead of using floating-point values. Another option is decimal floating-point arithmetic, as specified by ANSI/IEEE 754-2007. ISO/IEC WG14 has drafted a proposal to add support for decimal floating-point arithmetic to the C language [ISO/IEC DTR 24732].

When precise computation is necessary, carefully and methodically estimate the maximum cumulative error of the computations, regardless of whether decimal or binary is used, to ensure that the resulting error is within tolerances. Consider using numerical analysis to properly understand the problem. An introduction can be found in David Goldberg’s “What Every Computer Scientist Should Know about Floating-Point Arithmetic” [Goldberg 1991].

Noncompliant Code Example

This noncompliant code example takes the mean of 10 identical numbers and checks to see if the mean matches this number. It should match because the 10 numbers are all 10.1. Yet, because of the imprecision of floating-point arithmetic, the computed mean does not match this number.

```c
#include <stdio.h>

/* Returns the mean value of the array */
float mean(float array[], int size) {
    float total = 0.0;
    size_t i;
    for (i = 0; i < size; i++) {
        total += array[i];
        printf("array[%zu] = %f and total is %f\n", i, array[i], total);
    }
    if (size != 0)
        return total / size;
    else
        return 0.0;
}

enum { array_size = 10 };  
float array_value = 10.1;

#define _LIBCDF
#include "libcdf.h"

int main(void) {
    float array[array_size];
    float avg;
    size_t i;
    for (i = 0; i < array_size; i++) {
        array[i] = array_value;
    }
    avg = mean( array, array_size);
    printf("mean is %f\n", avg);
    if (avg == array[0]) {
        printf("array[0] is the mean\n");
    } else {
        printf("array[0] is not the mean\n");
    }
    return 0;
}
```

On a 64-bit Linux machine using GCC 4.1, this program yields the following output:
array[0] = 10.100000 and total is 10.100000
array[1] = 10.100000 and total is 20.200001
array[2] = 10.100000 and total is 30.300001
array[3] = 10.100000 and total is 40.400002
array[4] = 10.100000 and total is 50.500000
array[5] = 10.100000 and total is 60.599998
array[6] = 10.100000 and total is 70.699997
array[7] = 10.100000 and total is 80.799995
array[8] = 10.100000 and total is 90.899994
array[9] = 10.100000 and total is 100.999992
mean is 10.099999
array[0] is not the mean

Compliant Solution

The noncompliant code can be fixed by replacing the floating-point numbers with integers for the internal additions. Floats are used only when printing results and when doing the division to compute the mean.

```c
#include <stdio.h>

/* Returns the mean value of the array */
float mean(int array[], int size) {
    int total = 0;
    size_t i;
    for (i = 0; i < size; i++) {
        total += array[i];
        printf("array[%zu] = %f and total is %f\n", i, array[i] / 100.0, total / 100.0);
    }
    if (size != 0)
        return ((float) total) / size;
    else
        return 0.0;
}

enum {array_size = 10};
int array_value = 1010;

int main(void) {
    int array[array_size];
    float avg;
    size_t i;
    for (i = 0; i < array_size; i++) {
        array[i] = array_value;
    }
    avg = mean(array, array_size);
    printf("mean is %f\n", avg / 100.0);
    if (avg == array[0]) {
        printf("array[0] is the mean\n");
    } else {
        printf("array[0] is not the mean\n");
    }
    return 0;
}
```

On a 64-bit Linux machine using GCC 4.1, this program yields the following expected output:
Risk Assessment

Using a representation other than floating point may allow for more accurate results.

<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>FLP02-C</td>
<td>Low</td>
<td>Probable</td>
<td>High</td>
<td>P2</td>
<td>L3</td>
</tr>
</tbody>
</table>

Automated Detection

Checks for floating

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>20.10</td>
<td>float-comparison</td>
<td>Partially checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC-FLP02</td>
<td></td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td></td>
<td>Can detect violations of this recommendation. In particular, it checks to see if the arguments to an equality operator are of a floating-point type</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2021.2</td>
<td>C0790</td>
<td></td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>56 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++/test</td>
<td>2021.1</td>
<td>CERT_C-FLP02-a</td>
<td>Floating-point expressions shall not be tested for equality or inequality</td>
</tr>
<tr>
<td>PC-int Plus</td>
<td>1.4</td>
<td>777, 9252</td>
<td>Partially supported</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2021a</td>
<td>CERT C: Rec. FLP02-C</td>
<td>Checks for floating point comparison with equality operators (rec. partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0790</td>
<td></td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.13</td>
<td>V550</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>20.10</td>
<td>float-comparison</td>
<td>Partially checked</td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>FLP02-CPP. Avoid using floating-point numbers when precise computation is needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>NUM04-J. Do not use floating-point numbers if precise computation is required</td>
</tr>
<tr>
<td>ISO/IEC TR 24772:2013</td>
<td>Floating-point Arithmetic [PLF]</td>
</tr>
</tbody>
</table>

Bibliography
[Goldberg 1991]
[IEEE 754 2006]
[ISO/IEC DTR 24732]
[ISO/IEC JTC1/SC22/WG11]