**EXP43-C. Avoid undefined behavior when using restrict-qualified pointers**

An object that is accessed through a restrict-qualified pointer has a special association with that pointer. This association requires that all accesses to that object, directly or indirectly, use the value of that particular pointer. The intended use of the restrict qualifier is to promote optimization, and deleting all instances of the qualifier from a program does not change its meaning (that is, observable behavior). In the absence of this qualifier, other pointers can alias this object. Caching the value in an object designated through a restrict-qualified pointer is safe at the beginning of the block in which the pointer is declared because no preexisting aliases may also be used to reference that object. The cached value must be restored to the object by the end of the block, where preexisting aliases again become available. New aliases may be formed within the block, but these must all depend on the value of the restrict-qualified pointer so that they can be identified and adjusted to refer to the cached value. For a restrict-qualified pointer at file scope, the block is the body of each function in the file [Walls 2006]. Developers should be aware that C++ does not support the restrict qualifier, but some C++ compiler implementations support an equivalent qualifier as an extension.

The C Standard [ISO/IEC 9899:2011] identifies the following undefined behavior:

> A restrict-qualified pointer is assigned a value based on another restricted pointer whose associated block neither began execution before the block associated with this pointer, nor ended before the assignment (6.7.3.1).

This is an oversimplification, however, and it is important to review the formal definition of restrict in subclause 6.7.3.1 of the C Standard to properly understand undefined behaviors associated with the use of restrict-qualified pointers.

**Overlapping Objects**

The restrict qualifier requires that the pointers do not reference overlapping objects. If the objects referenced by arguments to functions overlap (meaning the objects share some common memory addresses), the behavior is undefined.

**Noncompliant Code Example**

This code example is noncompliant because an assignment is made between two restrict-qualified pointers in the same scope:

```c
int *restrict a;
int *restrict b;
extern int c[];

int main(void) {
    c[0] = 17;
    c[1] = 18;
    a = &c[0];
    b = &c[1];
    a = b; /* Undefined behavior */
    /* ... */
}
```

Note that undefined behavior occurs only when a is assigned to b. It is valid for a and b to point into the same array object, provided the range of elements accessed through one of the pointers does not overlap with the range of elements accessed through the other pointer.

**Compliant Solution**

One way to eliminate the undefined behavior is simply to remove the restrict-qualification from the affected pointers:

```c
int *a;
int *b;
extern int c[];

int main(void) {
    c[0] = 17;
    c[1] = 18;
    a = &c[0];
    b = &c[1];
    a = b; /* Defined behavior */
    /* ... */
}
```
restrict-Qualified Function Parameters

When calling functions that have restrict-qualified function parameters, it is important that the pointer arguments do not reference overlapping objects if one or more of the pointers are used to modify memory. Consequently, it is important to understand the semantics of the function being called.

Noncompliant Code Example

In this noncompliant code example, the function \( f() \) accepts three parameters. The function copies \( n \) integers from the int array referenced by the restrict-qualified pointer \( p \) to the int array referenced by the restrict-qualified pointer \( q \). Because the destination array is modified during each execution of the function (for which \( n \) is nonzero), if the array is accessed through one of the pointer parameters, it cannot also be accessed through the other. Declaring these function parameters as restrict-qualified pointers allows aggressive optimization by the compiler but can also result in undefined behavior if these pointers refer to overlapping objects.

```c
#include <stddef.h>
void f(size_t n, int *restrict p, const int *restrict q) {
    while (n-- > 0) {
        *p++ = *q++;
    }
}
void g(void) {
    extern int d[100];
    /* ... */
    f(50, d + 1, d); /* Undefined behavior */
}
```

The function \( g() \) declares an array \( d \) consisting of 100 int values and then invokes \( f() \) to copy memory from one area of the array to another. This call has undefined behavior because each of \( d[1] \) through \( d[49] \) is accessed through both \( p \) and \( q \).

Compliant Solution

In this compliant solution, the function \( f() \) is unchanged but the programmer has ensured that none of the calls to \( f() \) result in undefined behavior. The call to \( f() \) in \( g() \) is valid because the storage allocated to \( d \) is effectively divided into two disjoint objects.

```c
#include <stddef.h>
void f(size_t n, int *restrict p, const int *restrict q) {
    while (n-- > 0) {
        *p++ = *q++;
    }
}
void g(void) {
    extern int d[100];
    /* ... */
    f(50, d + 50, d); /* Defined behavior */
}
```

Noncompliant Code Example

In this noncompliant code example, the function \( add() \) adds the integer array referenced by the restrict-qualified pointers \( lhs \) to the integer array referenced by the restrict-qualified pointer \( rhs \) and stores the result in the restrict-qualified pointer referenced by \( res \). The function \( f() \) declares an array \( a \) consisting of 100 int values and then invokes \( add() \) to copy memory from one area of the array to another. The call \( add(100, a, a, a) \) has undefined behavior because the object modified by \( res \) is accessed by \( lhs \) and \( rhs \).
`#include <stddef.h>`

```c
void add(size_t n, int *restrict res, const int *restrict lhs, const int *restrict rhs) {
    for (size_t i = 0; i < n; ++i) {
        res[i] = lhs[i] + rhs[i];
    }
}
void f(void) {
    int a[100];
    add(100, a, a, a); /* Undefined behavior */
}
```

**Compliant Solution**

In this compliant solution, an unmodified object is aliased through two restricted pointers. Because `a` and `b` are disjoint arrays, a call of the form `add(100, a, b, b)` has defined behavior, because array `b` is not modified within function `add`.

```c
#include <stddef.h>

void add(size_t n, int *restrict res, const int *restrict lhs, const int *restrict rhs) {
    for (size_t i = 0; i < n; ++i) {
        res[i] = lhs[i] + rhs[i];
    }
}

void f(void) {
    int a[100];
    int b[100];
    add(100, a, b, b); /* Defined behavior */
}
```

**Invoking Library Functions with restrict-Qualified Pointers**

Ensure that `restrict`-qualified source and destination pointers do not reference overlapping objects when invoking library functions. For example, the following table lists C standard library functions that copy memory from a source object referenced by a `restrict`-qualified pointer to a destination object that is also referenced by a `restrict`-qualified pointer:

<table>
<thead>
<tr>
<th>Standard C</th>
<th>Annex K</th>
</tr>
</thead>
<tbody>
<tr>
<td>strcpy()</td>
<td><code>strcpy_s()</code></td>
</tr>
<tr>
<td><code>strncpy()</code></td>
<td><code>strncpy_s()</code></td>
</tr>
<tr>
<td>strcat()</td>
<td><code>strcat_s()</code></td>
</tr>
<tr>
<td><code>strncat()</code></td>
<td><code>strncat_s()</code></td>
</tr>
<tr>
<td><code>memcpy()</code></td>
<td><code>memmove()</code></td>
</tr>
<tr>
<td><code>strtok_s()</code></td>
<td><code>strtok_s()</code></td>
</tr>
</tbody>
</table>

If the objects referenced by arguments to functions overlap (meaning the objects share some common memory addresses), the behavior is **undefined**. (See also **undefined behavior**.) The result of the functions is unknown, and data may be corrupted. As a result, these functions must never be passed pointers to overlapping objects. If data must be copied between objects that share common memory addresses, a copy function guaranteed to work on overlapping memory, such as `memmove()`, should be used.

**Noncompliant Code Example**

In this noncompliant code example, the values of objects referenced by `ptr1` and `ptr2` become unpredictable after the call to `memcpy()` because their memory areas overlap:
```c
#include <string.h>

void func(void) {
    char c_str[] = "test string";
    char *ptr1 = c_str;
    char *ptr2;

    ptr2 = ptr1 + 3;
    /* Undefined behavior because of overlapping objects */
    memcpy(ptr2, ptr1, 6);
    /* ... */
}
```

**Compliant Solution**

In this compliant solution, the call to `memcpy()` is replaced with a call to `memmove()`. The `memmove()` function performs the same operation as `memcpy()` when the memory regions do not overlap. When the memory regions do overlap, the `n` characters from the object pointed to by the source (ptr1) are first copied into a temporary array of `n` characters that does not overlap the objects pointed to by the destination (ptr2) or the source. The `n` characters from the temporary array are then copied into the object pointed to by the destination.

```c
#include <string.h>

void func(void) {
    char c_str[] = "test string";
    char *ptr1 = c_str;
    char *ptr2;

    ptr2 = ptr1 + 3;
    memmove(ptr2, ptr1, 6);  /* Replace call to memcpy() */
    /* ... */
}
```

Similar solutions using `memmove()` can replace the string functions as long as care is taken regarding the byte size of the characters and proper null-termination of the copied string.

**Calling Functions with restrict-Qualified Pointer to a const-Qualified Type**

Ensure that functions that accept a `restrict`-qualified pointer to a `const`-qualified type do not modify the object referenced by that pointer. Formatted input and output standard library functions frequently fit this description. The following table lists some of the common functions for which the format argument is a `restrict`-qualified pointer to a `const`-qualified type.

<table>
<thead>
<tr>
<th>Standard C</th>
<th>Annex K</th>
</tr>
</thead>
<tbody>
<tr>
<td>printf()</td>
<td>printf_s()</td>
</tr>
<tr>
<td>scanf()</td>
<td>scanf_s()</td>
</tr>
<tr>
<td>sprintf()</td>
<td>sprintf_s()</td>
</tr>
<tr>
<td>snprintf()</td>
<td>snprintf_s()</td>
</tr>
</tbody>
</table>

For formatted output functions such as `printf()`, it is unlikely that a programmer would modify the format string. However, an attacker may attempt to do so if a program violates FIO30-C. Exclude user input from format strings and passes tainted values as part of the format string.

**Noncompliant Code Example**

In this noncompliant code example, the programmer is attempting to overwrite the format string with a string value read in from `stdin` such as "%d%f 1 3.3" and use the resulting modified string of "%s%f" to input the subsequent values of 1 and 3.3.
#include <stdio.h>

void func(void) {
    int i;
    float x;
    char format[100] = "%s";
    /* Undefined behavior */
    int n = scanf(format, format + 2, &i, &x);
    /* ... */
}

Compliant Solution

The intended results are achieved by this compliant solution:

```c
#include <stdio.h>

void func(void) {
    int i;
    float x;
    int n = scanf("%d%f", &i, &x); /* Defined behavior */
    /* ... */
}
```

Outer-to-Inner Assignments between Restricted Pointers

The assignment between restrict-qualified pointers declared in an inner nested block from an outer block has defined behavior.

Noncompliant Code Example

The assignment of restrict-qualified pointers to other restrict-qualified pointers within the same block has undefined behavior:

```c
void func(void) {
    int *restrict p1;
    int *restrict q1;

    int *restrict p2 = p1; /* Undefined behavior */
    int *restrict q2 = q1; /* Undefined behavior */
}
```

Compliant Solution

The intended results can be achieved using an inner nested block, as shown in this compliant solution:

```c
void func(void) {
    int *restrict p1;
    int *restrict q1;
    {
        /* Added inner block */
        int *restrict p2 = p1; /* Valid, well-defined behavior */
        int *restrict q2 = q1; /* Valid, well-defined behavior */
    }
}
```

Risk Assessment

The incorrect use of restrict-qualified pointers can result in undefined behavior that might be exploited to cause data integrity violations.

<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>EXP43-C</td>
<td>Medium</td>
<td>Probable</td>
<td>High</td>
<td>P4</td>
<td>L3</td>
</tr>
</tbody>
</table>
Related Vulnerabilities

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

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>LANG.TYPE.RESTRICT</td>
<td>Restrict qualifier used</td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>MISRA C 2012 Rule 8.14</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>GCC</td>
<td>8.1</td>
<td>-Wrestrict</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>480 S, 489 S, 613 S</td>
<td>Enhanced enforcement</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT.C-EXP43-a</td>
<td>The restrict type qualifier shall not be used</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule EXP43-C</td>
<td>Checks for copy of overlapping memory (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>1057</td>
<td></td>
</tr>
<tr>
<td>SonarQube C/C++ Plugin</td>
<td>3.11</td>
<td>S1836</td>
<td>Implements MISRA C:2012 Rule 8.14 to flag uses of restrict</td>
</tr>
</tbody>
</table>

Related Guidelines

Key here (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>FIO30-C. Exclude user input from format strings</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>ISO/IEC TS 17961</td>
<td>Passing pointers into the same object as arguments to different restrict-qualified parameters [restrict]</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>MISRA C:2012</td>
<td>Rule 8.14 (required)¹</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
</tbody>
</table>

¹  MISRA Rule 8.14 prohibits the use of the restrict keyword except in C standard library functions.

Bibliography

[ISO/IEC 9899:2011] 6.7.3.1, "Formal Definition of restrict"

[Walls 2006]