EXP39-C. Do not access a variable through a pointer of an incompatible type

Modifying a variable through a pointer of an incompatible type (other than unsigned char) can lead to unpredictable results. Subclause 6.2.7 of the C Standard states that two types may be distinct yet compatible and addresses precisely when two distinct types are compatible.

This problem is often caused by a violation of aliasing rules. The C Standard, 6.5, paragraph 7 [ISO/IEC 9899:2011], specifies those circumstances in which an object may or may not be aliased.

An object shall have its stored value accessed only by an lvalue expression that has one of the following types:

- a type compatible with the effective type of the object,
- a qualified version of a type compatible with the effective type of the object,
- a type that is the signed or unsigned type corresponding to the effective type of the object,
- an aggregate or union type that includes one of the aforementioned types among its members (including, recursively, a member of a subaggregate or contained union), or
- a character type.

Accessing an object by means of any other lvalue expression (other than unsigned char) is undefined behavior 37.

Noncompliant Code Example

In this noncompliant example, an object of type float is incremented through an int *. The programmer can use the unit in the last place to get the next representable value for a floating-point type. However, accessing an object through a pointer of an incompatible type is undefined behavior.

```c
#include <stdio.h>

void f(void) {
    if (sizeof(int) == sizeof(float)) {
        float f = 0.0f;
        int *ip = (int *)&f;
        (*ip)++;
        printf("float is %f\n", f);
    }
}
```

Compliant Solution

In this compliant solution, the standard C function nextafterf() is used to round toward the highest representable floating-point value:

```c
#include <float.h>
#include <math.h>
#include <stdio.h>

void f(void) {
    float f = 0.0f;
    f = nextafterf(f, FLT_MAX);
    printf("float is %f\n", f);
}
```

Noncompliant Code Example

In this noncompliant code example, an array of two values of type short is treated as an integer and assigned an integer value. The resulting values are indeterminate.
```c
#include <stdio.h>

void func(void) {
    short a[2];
    a[0]=0x1111;
    a[1]=0x1111;

    *(int *)a = 0x22222222;

    printf("%x %x\n", a[0], a[1]);
}
```

When translating this code, an implementation can assume that no access through an integer pointer can change the array `a`, consisting of shorts. Consequently, `printf()` may be called with the original values of `a[0]` and `a[1].`

### Implementation Details

Recent versions of GCC turn on the option `-fstrict-aliasing`, which allows alias-based optimizations, by default with `-O2`. Some architectures then print "1111 1111" as a result. Without optimization, the executable generates the expected output "2222 2222."

To disable optimizations based on alias analysis for faulty legacy code, the option `-fno-strict-aliasing` can be used as a workaround. The option `-Wstrict-aliasing`, which is included in `-Wall`, warns about some, but not all, violations of aliasing rules when `-fstrict-aliasing` is active.

When GCC 3.4.6 compiles this code with optimization, the assignment through the aliased pointer is effectively eliminated.

### Compliant Solution

This compliant solution uses a `union` type that includes a type compatible with the effective type of the object:

```c
#include <stdio.h>

void func(void) {
    union {
        short a[2];
        int i;
    } u;

    u.a[0]=0x1111;
    u.a[1]=0x1111;
    u.i = 0x22222222;

    printf("%x %x\n", u.a[0], u.a[1]);
    /* ... */
}
```

The `printf()` behavior in this compliant solution is unspecified, but it is commonly accepted as an implementation extension. (See unspecified behavior 11.)

This function typically outputs "2222 2222." However, there is no guarantee that this will be true, even on implementations that defined the unspecified behavior; values of type `short` need not have the same representation as values of type `int`.

### Noncompliant Code Example

In this noncompliant code example, a `gadget` object is allocated, then `realloc()` is called to create a `widget` object using the memory from the `gadget` object. Although reusing memory to change types is acceptable, accessing the memory copied from the original object is undefined behavior.
#include <stdlib.h>

struct gadget {
    int i;
    double d;
    char *p;
};

struct widget {
    char *q;
    int j;
    double e;
};

void func(void) {
    struct gadget *gp;
    struct widget *wp;

    gp = (struct gadget *)malloc(sizeof(struct gadget));
    if (!gp) {
        /* Handle error */
    }
    /* ... Initialize gadget ... */
    wp = (struct widget *)realloc(gp, sizeof(struct widget));
    if (!wp) {
        free(gp);
        /* Handle error */
    }
    if (wp->j == 12) {
        /* ... */
    }
    /* ... */
    free(wp);
}

Compliant Solution

This compliant solution reuses the memory from the gadget object but reinitializes the memory to a consistent state before reading from it:
#include <stdlib.h>
#include <string.h>

struct gadget {
    int i;
    double d;
    char *p;
};

struct widget {
    char *q;
    int j;
    double e;
};

void func(void) {
    struct gadget *gp;
    struct widget *wp;

    gp = (struct gadget *)malloc(sizeof (struct gadget));
    if (!gp) {
        /* Handle error */
        /* ... */
    }
    /* ... */
    wp = (struct widget *)realloc(gp, sizeof(struct widget));
    if (!wp) {
        free(gp);
        /* Handle error */
    }
    memset(wp, 0, sizeof(struct widget));
    /* ... Initialize widget ... */

    if (wp->j == 12) {
        /* ... */
    }
    /* ... */
    free(wp);
}

Noncompliant Code Example

According to the C Standard, 6.7.6.2 [ISO/IEC 9899:2011], using two or more incompatible arrays in an expression is undefined behavior. (See also undefined behavior 76.)

For two array types to be compatible, both should have compatible underlying element types, and both size specifiers should have the same constant value. If either of these properties is violated, the resulting behavior is undefined.

In this noncompliant code example, the two arrays \( a \) and \( b \) fail to satisfy the equal size specifier criterion for array compatibility. Because \( a \) and \( b \) are not equal, writing to what is believed to be a valid member of \( a \) might exceed its defined memory boundary, resulting in an arbitrary memory overwrite.

```c
enum { ROWS = 10, COLS = 15 }

void func(void) {
    int a[ROWS][COLS];
    int (*b)[ROWS] = a;
}
```

Most compilers will produce a warning diagnostic if the two array types used in an expression are incompatible.

Compliant Solution

In this compliant solution, \( b \) is declared to point to an array with the same number of elements as \( a \), satisfying the size specifier criterion for array compatibility:
enum { ROWS = 10, COLS = 15 };  

void func(void) {
    int a[ROWS][COLS];
    int (*b)[COLS] = a;
}

Risk Assessment

Optimizing for performance can lead to aliasing errors that can be quite difficult to detect. Furthermore, as in the preceding example, unexpected results can lead to buffer overflow attacks, bypassing security checks, or unexpected execution.

<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>EXP39-C</td>
<td>Medium</td>
<td>Unlikely</td>
<td>High</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>94 S, 554 S</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-EXP39-a</td>
<td>There shall be no implicit conversions from integral to floating type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-EXP39-b</td>
<td>A cast should not be performed between a pointer to object type and a different pointer to object type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-EXP39-c</td>
<td>Avoid accessing arrays and pointers out of bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-EXP39-d</td>
<td>Avoid buffer overflow from tainted data due to defining incorrect format limits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-EXP39-e</td>
<td>Avoid buffer read overflow from tainted data</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-EXP39-f</td>
<td>Avoid buffer write overflow from tainted data</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule EXP39-C</td>
<td>Checks for cast to pointer pointing to object of different type (rule partially covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0310, 0751, 3305</td>
<td>Partially implemented</td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>3017, 3030, 3033</td>
<td></td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.07</td>
<td>V580</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

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

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>ISO/IEC TS 17961</td>
<td>Accessing an object through a pointer to an incompatible type [ptrcomp]</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-119, Improper Restriction of Operations within the Bounds of a Memory Buffer</td>
<td>2017-05-18: CERT: Partial overlap</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-125, Out-of-bounds Read</td>
<td>2017-05-18: CERT: Partial overlap</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-704</td>
<td>2017-06-14: CERT: Rule subset of CWE</td>
</tr>
</tbody>
</table>

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-119 and EXP39-C


Intersection( EXP39-C, CWE-119) =
• Reading memory assigned to one type, but being accessed through a pointer to a larger type.

EXP39-C – CWE-119 =

• Writing to memory assigned to one type, but accessed through a pointer to a larger type

• Reading memory assigned to one type, but being accessed through a pointer to a smaller (or equal-sized) type

CWE-119 – EXP39-C =

• Reading beyond a buffer using a means other than accessing a variable through an incompatible pointer.

CWE-123 and EXP39-C

Intersection( CWE-123, EXP39-C) = ∅

EXP39-C allows overflowing a (small) buffer, but not arbitrary memory writes. (Possibly an arbitrary-memory write exploit could be devised using a “perfect storm” of incompatible types, but this would be uncommon in practice.)

CWE-125 and EXP39-C

Independent( ARR30-C, ARR38-C, EXP39-C, INT30-C) STR31-C = Subset( Union( ARR30-C, ARR38-C)) STR32-C = Subset( ARR38-C)

Intersection( EXP39-C, CWE-125) =

• Reading memory assigned to one type, but being accessed through a pointer to a larger type.

ESP39-C – CWE-125 =

• Reading memory assigned to one type, but being accessed through a pointer to a smaller (or equal-sized) type

CWE-125 – EXP39-C =

• Reading beyond a buffer using a means other than accessing a variable through an incompatible pointer.

CWE-188 and EXP39-C

Intersection( CWE-188, EXP39-C) = ∅

CWE-188 appears to be about making assumptions about the layout of memory between distinct variables (that are not part of a larger struct or array). Such assumptions typically involve pointer arithmetic (which violates ARR30-C). EXP39-C involves only one object in memory being (incorrectly) interpreted as if it were another object. EG a float being treated as an int (usually via pointers and typecasting)

CWE-704 and EXP39-C

CWE-704 = Union( EXP39-C, list) where list =

• Incorrect (?) typecast that is not incompatible
### Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acton 2006</td>
<td>&quot;Understanding Strict Aliasing&quot;</td>
</tr>
<tr>
<td>GCC Known Bugs</td>
<td>&quot;C Bugs, Aliasing Issues while Casting to Incompatible Types&quot;</td>
</tr>
<tr>
<td></td>
<td>6.7.6.2, &quot;Array Declarators&quot;</td>
</tr>
<tr>
<td>Walfridsson 2003</td>
<td>Aliasing, Pointer Casts and GCC 3.3</td>
</tr>
</tbody>
</table>