ARR30-C. Do not form or use out-of-bounds pointers or array subscripts

The C Standard identifies the following distinct situations in which undefined behavior (UB) can arise as a result of invalid pointer operations:

<table>
<thead>
<tr>
<th>UB</th>
<th>Description</th>
<th>Example Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>46</td>
<td>Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that does not point into, or just beyond, the same array object.</td>
<td>Forming Out-of-Bounds Pointer, Null Pointer Arithmetic</td>
</tr>
<tr>
<td>47</td>
<td>Addition or subtraction of a pointer into, or just beyond, an array object and an integer type produces a result that points just beyond the array object and is used as the operand of a unary * operator that is evaluated.</td>
<td>Dereferencing Past the End Pointer, Using Past the End Index</td>
</tr>
<tr>
<td>49</td>
<td>An array subscript is out of range, even if an object is apparently accessible with the given subscript, for example, in the lvalue expression a[i][j] given the declaration int a[4][5].</td>
<td>Apparently Accessible Out-of-Range Index</td>
</tr>
<tr>
<td>62</td>
<td>An attempt is made to access, or generate a pointer to just past, a flexible array member of a structure when the referenced object provides no elements for that array.</td>
<td>Pointer Past Flexible Array Member</td>
</tr>
</tbody>
</table>

Noncompliant Code Example (Forming Out-of-Bounds Pointer)

In this noncompliant code example, the function `f()` attempts to validate the `index` before using it as an offset to the statically allocated `table` of integers. However, the function fails to reject negative `index` values. When `index` is less than zero, the behavior of the addition expression in the return statement of the function is undefined behavior 46. On some implementations, the addition alone can trigger a hardware trap. On other implementations, the addition may produce a result that when dereferenced triggers a hardware trap. Other implementations still may produce a dereferenceable pointer that points to an object distinct from `table`. Using such a pointer to access the object may lead to information exposure or cause the wrong object to be modified.

```c
enum { TABLESIZE = 100 };
static int table[TABLESIZE];

int *f(int index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

Compliant Solution

One compliant solution is to detect and reject invalid values of `index` if using them in pointer arithmetic would result in an invalid pointer:

```c
enum { TABLESIZE = 100 };  
static int table[TABLESIZE];

int *f(int index) {
    if (index >= 0 && index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}
```

Compliant Solution

Another slightly simpler and potentially more efficient compliant solution is to use an unsigned type to avoid having to check for negative values while still rejecting out-of-bounds positive values of `index`:
#include <stddef.h>
enum { TABLESIZE = 100 };
static int table[TABLESIZE];
int *f(size_t index) {
    if (index < TABLESIZE) {
        return table + index;
    }
    return NULL;
}

Noncompliant Code Example (Dereferencing Past-the-End Pointer)

This noncompliant code example shows the flawed logic in the Windows Distributed Component Object Model (DCOM) Remote Procedure Call (RPC) interface that was exploited by the W32.Blaster.Worm. The error is that the while loop in the GetMachineName() function (used to extract the host name from a longer string) is not sufficiently bounded. When the character array pointed to by pwszTemp does not contain the backslash character among the first MAX_COMPUTERNAME_LENGTH_FQDN + 1 elements, the final valid iteration of the loop will dereference past the end pointer, resulting in exploitable undefined behavior 47. In this case, the actual exploit allowed the attacker to inject executable code into a running program. Economic damage from the Blaster worm has been estimated to be at least $525 million [Pethia 2003].

For a discussion of this programming error in the Common Weakness Enumeration database, see CWE-119, "Improper Restriction of Operations within the Bounds of a Memory Buffer," and CWE-121, "Stack-based Buffer Overflow" [MITRE 2013].

Compliant Solution

In this compliant solution, the while loop in the GetMachineName() function is bounded so that the loop terminates when a backslash character is found, the null-termination character (L'\0') is discovered, or the end of the buffer is reached. Or, as coded, the while loop continues as long as each character is neither a backslash nor a null character and is not at the end of the buffer. This code does not result in a buffer overflow even if no backslash character is found in wszMachineName.
HRESULT GetMachineName(
    wchar_t *pwszPath,
    wchar_t wszMachineName[MAX_COMPUTERNAME_LENGTH_FQDN+1])
{
    wchar_t *pwszServerName = wszMachineName;
    wchar_t *pwszTemp = pwszPath + 2;
    wchar_t *end_addr = pwszServerName + MAX_COMPUTERNAME_LENGTH_FQDN;
    while ((*pwszTemp != L'\') &&
        (*pwszTemp != L'\0') &&
        (pwszServerName < end_addr))
    {
        *pwszServerName++ = *pwszTemp++;
    }
    /* ... */
}

This compliant solution is for illustrative purposes and is not necessarily the solution implemented by Microsoft. This particular solution may not be correct because there is no guarantee that a backslash is found.

Noncompliant Code Example (Using Past-the-End Index)

Similar to the dereferencing-past-the-end-pointer error, the function insert_in_table() in this noncompliant code example uses an otherwise valid index to attempt to store a value in an element just past the end of an array.

First, the function incorrectly validates the index pos against the size of the buffer. When pos is initially equal to size, the function attempts to store value in a memory location just past the end of the buffer.

Second, when the index is greater than size, the function modifies size before growing the size of the buffer. If the call to realloc() fails to increase the size of the buffer, the next call to the function with a value of pos equal to or greater than the original value of size will again attempt to store value in a memory location just past the end of the buffer or beyond.

Third, the function violates INT30-C. Ensure that unsigned integer operations do not wrap, which could lead to wrapping when 1 is added to pos or when size is multiplied by the size of int.

For a discussion of this programming error in the Common Weakness Enumeration database, see CWE-122, "Heap-based Buffer Overflow," and CWE-129, "Improper Validation of Array Index" [MITRE 2013].

#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size < pos) {
        int *tmp;
        size = pos + 1;
        tmp = (int *)realloc(table, sizeof(*table) * size);
        if (tmp == NULL) {
            return -1;   /* Failure */
        }
        table = tmp;
    }

    table[pos] = value;
    return 0;
}

Compliant Solution

This compliant solution correctly validates the index pos by using the <= relational operator, ensures the multiplication will not overflow, and avoids modifying size until it has verified that the call to realloc() was successful:
```c
#include <stdint.h>
#include <stdlib.h>

static int *table = NULL;
static size_t size = 0;

int insert_in_table(size_t pos, int value) {
    if (size <= pos) {
        if ((SIZE_MAX - 1 < pos) ||
            ((pos + 1) > SIZE_MAX / sizeof(*table))) {
            return -1;
        }

        int *tmp = (int *)realloc(table, sizeof(*table) * (pos + 1));
        if (tmp == NULL) {
            return -1;
        }

        /* Modify size only after realloc() succeeds */
        size = pos + 1;
        table = tmp;
    }

    table[pos] = value;
    return 0;
}
```

**Noncompliant Code Example (Apparently Accessible Out-of-Range Index)**

This noncompliant code example declares `matrix` to consist of 7 rows and 5 columns in row-major order. The function `init_matrix` iterates over all 35 elements in an attempt to initialize each to the value given by the function argument `x`. However, because multidimensional arrays are declared in C in row-major order, the function iterates over the elements in column-major order, and when the value of `j` reaches the value `COLS` during the first iteration of the outer loop, the function attempts to access element `matrix[0][5]`. Because the type of `matrix` is `int[7][5]`, the `j` subscript is out of range, and the access has undefined behavior.

```c
#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < COLS; i++) {
        for (size_t j = 0; j < ROWS; j++) {
            matrix[i][j] = x;
        }
    }
}
```

**Compliant Solution**

This compliant solution avoids using out-of-range indices by initializing `matrix` elements in the same row-major order as multidimensional objects are declared in C:

```c
#include <stddef.h>
#define COLS 5
#define ROWS 7
static int matrix[ROWS][COLS];

void init_matrix(int x) {
    for (size_t i = 0; i < ROWS; i++) {
        for (size_t j = 0; j < COLS; j++) {
            matrix[i][j] = x;
        }
    }
}
```
Noncompliant Code Example (Pointer Past Flexible Array Member)

In this noncompliant code example, the function `find()` attempts to iterate over the elements of the flexible array member `buf`, starting with the second element. However, because function `g()` does not allocate any storage for the member, the expression `first++` in `find()` attempts to form a pointer just past the end of `buf` when there are no elements. This attempt is undefined behavior. (See MSC21-C. Use robust loop termination conditions for more information.)

```c
#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last  = s->buf + s->len;

    while (first++ != last) { /* Undefined behavior */
        if (*first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}
```

Compliant Solution

This compliant solution avoids incrementing the pointer unless a value past the pointer's current value is known to exist:

```c
#include <stdlib.h>

struct S {
    size_t len;
    char buf[]; /* Flexible array member */
};

const char *find(const struct S *s, int c) {
    const char *first = s->buf;
    const char *last  = s->buf + s->len;

    while (first != last) { /* Avoid incrementing here */
        if (*++first == (unsigned char)c) {
            return first;
        }
    }
    return NULL;
}

void g(void) {
    struct S *s = (struct S *)malloc(sizeof(struct S));
    if (s == NULL) {
        /* Handle error */
    }
    s->len = 0;
    find(s, 'a');
}
```
Noncompliant Code Example (Null Pointer Arithmetic)

This noncompliant code example is similar to an Adobe Flash Player vulnerability that was first exploited in 2008. This code allocates a block of memory and initializes it with some data. The data does not belong at the beginning of the block, which is left uninitialized. Instead, it is placed offset bytes within the block. The function ensures that the data fits within the allocated block.

```c
#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset, char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```

This function fails to check if the allocation succeeds, which is a violation of ERR33-C. Detect and handle standard library errors. If the allocation fails, then malloc() returns a null pointer. The null pointer is added to offset and passed as the destination argument to memcpy(). Because a null pointer does not point to a valid object, the result of the pointer arithmetic is undefined behavior.

An attacker who can supply the arguments to this function can exploit it to execute arbitrary code. This can be accomplished by providing an overly large value for block_size, which causes malloc() to fail and return a null pointer. The offset argument will then serve as the destination address to the call to memcpy(). The attacker can specify the data and data_size arguments to provide the address and length of the address, respectively, that the attacker wishes to write into the memory referenced by offset. The overall result is that the call to memcpy() can be exploited by an attacker to overwrite an arbitrary memory location with an attacker-supplied address, typically resulting in arbitrary code execution.

Compliant Solution (Null Pointer Arithmetic)

This compliant solution ensures that the call to malloc() succeeds:

```c
#include <string.h>
#include <stdlib.h>

char *init_block(size_t block_size, size_t offset, char *data, size_t data_size) {
    char *buffer = malloc(block_size);
    if (NULL == buffer) {
        /* Handle error */
    }
    if (data_size > block_size || block_size - data_size < offset) {
        /* Data won't fit in buffer, handle error */
    }
    memcpy(buffer + offset, data, data_size);
    return buffer;
}
```

Risk Assessment

Writing to out-of-range pointers or array subscripts can result in a buffer overflow and the execution of arbitrary code with the permissions of the vulnerable process. Reading from out-of-range pointers or array subscripts can result in unintended information disclosure.

<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>ARR30-C</td>
<td>High</td>
<td>Likely</td>
<td>High</td>
<td>P9</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection
<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astrée</td>
<td>22.04</td>
<td>array-index-range, array-index-range-constant, null-dereferencing, pointed-deallocation, return-reference-local</td>
<td>Partially checked. Can detect all accesses to invalid pointers as well as array index out-of-bounds accesses and prove their absence. This rule is only partially checked as invalid but unused pointers may not be reported.</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>7.2.0</td>
<td>CertC-ARR30</td>
<td>Can detect out-of-bound access to array / buffer.</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.1p0</td>
<td>LANG.MEM.BO, LANG.MEM.BU, LANG.MEM.TBA, LANG.MEM.TO, LANG.MEM.TU, LANG.STRUCT.PARITH, LANG.STRUCT.PBB, LANG.STRUCT.PPE, BADFUNC.BO.*</td>
<td>Buffer overrun, Buffer underrun, Tainted buffer access, Type overrun, Type underrun, Pointer Arithmetic, Pointer before beginning of object, Pointer past end of object. A collection of warning classes that report uses of library functions prone to internal buffer overflows.</td>
</tr>
<tr>
<td>Compass /ROSE</td>
<td></td>
<td></td>
<td>Could be configured to catch violations of this rule. The way to catch the noncompliant code example is to first hunt for example code that follows this pattern: for (LPWSTR pwszTemp = pwszPath + 2; *pwszTemp != L''; *pwszTemp++); In particular, the iteration variable is a pointer, it gets incremented, and the loop condition does not set an upper bound on the pointer. Once this case is handled, ROSE can handle cases like the real noncompliant code example, which is effectively the same semantics, just different syntax.</td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>OVERRUN, NEGATIVE_RETURNS, ARRAY_VS_SINGLETON, BUFFER_SIZE</td>
<td>Can detect the access of memory past the end of a memory buffer/array. Can detect when the loop bound may become negative. Can detect the out-of-bound read/write to array allocated statically or dynamically. Can detect buffer overflows.</td>
</tr>
<tr>
<td>Cppcheck</td>
<td>1.66</td>
<td>arrayIndexOutOfBounds, outOfBounds, negativeIndex, arrayIndexThenCheck, arrayIndexOutOfBoundsCond, possibleBufferAccessOutOfBounds</td>
<td>Context sensitive analysis of array index, pointers, etc. Array index out of bounds. Buffer overflow when calling various functions memset, strcpy,... Warns about condition (a[i] == 0 &amp;&amp; i &lt; unknown_value) and recommends that (i &lt; unknown_value &amp;&amp; a[i] == 0) is used instead. Detects unsafe code when array is accessed before/after it is tested if the array index is out of bounds.</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.2</td>
<td>C2820, C2821, C2822, C2823, C2840, C2841, C2842, C2843, C2930, C2931, C2932, C2933, C2935, C2936, C2937, C2938, C2950, C2951, C2952, C2953, C++2820, C++2821, C++2822, C++2823, C++2840, C++2841, C++2842, C++2843, C++2930, C++2931, C++2932, C++2933, C++2935, C++2936, C++2937, C++2938, C++2950, C++2951, C++2952, C++2953</td>
<td>Detects unsafe code when array is accessed before/after it is tested if the array index is out of bounds.</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2022.2</td>
<td>ABV.GENERAL, ABV.GENERAL.MULTIDIMENSION, NPD.FUNC.CALL.MIGHT, ABV.ANY_SIZE_ARRAY, ABV.STACK, ABV.TAINTED, ABV.UNICODE.BOUND_MAP, ABV.UNICODE.FAILED_MAP, ABV.UNICODE.NNTS_MAP, ABV.UNICODE.SELF_MAP, ABV.UNKNOW_SIZE, NNTS.MIGHT, NNTS.MUST, NNTS.TAINTED, SV.TAINTED.INDICES_ACCESS, SV.TAINTED.LOOP_BOUND</td>
<td>Partially implemented.</td>
</tr>
</tbody>
</table>
Related Vulnerabilities

CVE-2008-1517 results from a violation of this rule. Before Mac OSX version 10.5.7, the XNU kernel accessed an array at an unverified user-input index, allowing an attacker to execute arbitrary code by passing an index greater than the length of the array and therefore accessing outside memory [xorl 2009].

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://csrc.nist.gov/Directory/cert/vuln/).

STR31-C = Subset( Union( ARR30-C, ARR38-C))
STR32-C = Subset( ARR38-C)
CWE-119 = Union( ARR30-C, ARR38-C)
Intersection( ARR30-C, ARR38-C) = Ø

CWE-394 and ARR30-C
Intersection( ARR30-C, CWE-394) = Ø
CWE-394 deals with potentially-invalid function return values. Which may be used as an (invalid) array index, but validating the return value is a separate operation.

CWE-125 and ARR30-C
Independent( ARR30-C, ARR38-C, EXP39-C, INT30-C)
STR31-C = Subset( Union( ARR30-C, ARR38-C))
STR32-C = Subset( ARR38-C)
CWE-125 = Subset( CWE-119) = Union( ARR30-C, ARR38-C)
Intersection( ARR30-C, CWE-125) =
  • Reading from an out-of-bounds array index, or off the end of an array
ARR30-C – CWE-125 =
  • Writing to an out-of-bounds array index, or off the end of an array
CWE-125 – ARR30-C =
  • Reading beyond a non-array buffer
  • Using a library function to achieve an out-of-bounds read.

CWE-123 and ARR30-C
Independent( ARR30-C, ARR38-C)
STR31-C = Subset( Union( ARR30-C, ARR38-C))
STR32-C = Subset( ARR38-C)
Intersection( CWE-123, ARR30-C) =
  • Write of arbitrary value to arbitrary (probably invalid) array index
ARR30-C – CWE-123 =
  • Read of value from arbitrary (probably invalid) array index
  • Construction of invalid index (pointer arithmetic)
CWE-123 – ARR30-C =
  • Arbitrary writes that do not involve directly constructing an invalid array index

CWE-129 and ARR30-C
Independent( ARR30-C, ARR32-C, INT31-C, INT32-C)
ARR30-C = Union( CWE-129, list), where list =
  • Dereferencing an out-of-bounds array index, where index is a trusted value
  • Forming an out-of-bounds array index, without dereferencing it, whether or not index is a trusted value. (This excludes the array’s TOOFAR index, which is one past the final element; this behavior is well-defined in C11.)

CWE-120 and ARR30-C
See CWE-120 and MEM35-C

**CWE-122 and ARR30-C**

Intersection( ARR30-C, CWE-122) = Ø

CWE-122 specifically addresses buffer overflows on the heap operations, which occur in the context of string-copying. ARR30 specifically addresses improper creation or references of array indices. Which might happen as part of a heap buffer overflow, but is on a lower programming level.

**CWE-20 and ARR30-C**

See CWE-20 and ERR34-C

**CWE-687 and ARR30-C**

Intersection( CWE-687, ARR30-C) = Ø

ARR30-C is about invalid array indices which are created through pointer arithmetic, and dereferenced through an operator (* or []). Neither involve function calls, thus CWE-687 does not apply.

**CWE-786 and ARR30-C**

ARR30-C = Union( CWE-786, list) where list =

- Access of memory location after end of buffer
- Construction of invalid array reference (pointer). This does not include an out-of-bounds array index (an integer).

**CWE-789 and ARR30-C**

Intersection( CWE-789, ARR30-C) = Ø

CWE-789 is about allocating memory, not array subscripting

**Bibliography**

| [Finlay 2003] |  |
| [Microsoft 2003] |  |
| [Pethia 2003] |  |
| [Seacord 2013b] | Chapter 1, "Running with Scissors" |
| [Viega 2005] | Section 5.2.13, "Unchecked Array Indexing" |
| [xorl 2009 ] | "CVE-2008-1517: Apple Mac OS X (XNU) Missing Array Index Validation" |