MEM35-C. Allocate sufficient memory for an object

The types of integer expressions used as size arguments to malloc(), calloc(), realloc(), or aligned_alloc() must have sufficient range to represent the size of the objects to be stored. If size arguments are incorrect or can be manipulated by an attacker, then a buffer overflow may occur. Incorrect size arguments, inadequate range checking, integer overflow, or truncation can result in the allocation of an inadequately sized buffer.

Typically, the amount of memory to allocate will be the size of the type of object to allocate. When allocating space for an array, the size of the object will be multiplied by the bounds of the array. When allocating space for a structure containing a flexible array member, the size of the array member must be added to the size of the structure. (See MEM33-C. Allocate and copy structures containing a flexible array member dynamically.) Use the correct type of the object when computing the size of memory to allocate.

STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator is a specific instance of this rule.

Noncompliant Code Example (Pointer)

In this noncompliant code example, inadequate space is allocated for a struct tm object because the size of the pointer is being used to determine the size of the pointed-to object:

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

struct tm *make_tm(int year, int mon, int day, int hour,
                    int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}
```

Compliant Solution (Pointer)

In this compliant solution, the correct amount of memory is allocated for the struct tm object. When allocating space for a single object, passing the (dereferenced) pointer type to the operator is a simple way to allocate sufficient memory. Because the operator does not evaluate its operand, dereferencing an uninitialized or null pointer in this context is well-defined behavior.

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

struct tm *make_tm(int year, int mon, int day, int hour,
                    int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(*tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}
```

Noncompliant Code Example (Integer)

In this noncompliant code example, an array of long is allocated and assigned to p. The code attempts to check for unsigned integer overflow in compliance with INT30-C. Ensure that unsigned integer operations do not wrap and also ensures that len is not equal to zero. (See MEM04-C. Beware of zero-length allocations.) However, because sizeof(int) is used to compute the size, and not sizeof(long), an insufficient amount of memory can be allocated on implementations where sizeof(long) is larger than sizeof(int), and filling the array can cause a heap buffer overflow.

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

struct tm *make_tm(int year, int mon, int day, int hour,
                    int min, int sec) {
    struct tm *tmb;
    tmb = (struct tm *)malloc(sizeof(tmb));
    if (tmb == NULL) {
        return NULL;
    }
    *tmb = (struct tm) {
        .tm_sec = sec, .tm_min = min, .tm_hour = hour,
        .tm_mday = day, .tm_mon = mon, .tm_year = year
    };
    return tmb;
}
```
```c
#include <stdint.h>
#include <stdlib.h>

void function(size_t len) {
  long *p;
  if (len == 0 || len > SIZE_MAX / sizeof(long)) {
    /* Handle overflow */
  }
  p = (long *)malloc(len * sizeof(long));
  if (p == NULL) {
    /* Handle error */
  }
  free(p);
}
```

### Compliant Solution (Integer)

This compliant solution uses `sizeof(long)` to correctly size the memory allocation:

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

void function(size_t len) {
  long *p;
  if (len == 0 || len > SIZE_MAX / sizeof(long)) {
    /* Handle overflow */
  }
  p = (long *)malloc(len * sizeof(long));
  if (p == NULL) {
    /* Handle error */
  }
  free(p);
}
```

### Compliant Solution (Integer)

Alternatively, `sizeof(*p)` can be used to properly size the allocation:

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

void function(size_t len) {
  long *p;
  if (len == 0 || len > SIZE_MAX / sizeof(*p)) {
    /* Handle overflow */
  }
  p = (long *)malloc(len * sizeof(*p));
  if (p == NULL) {
    /* Handle error */
  }
  free(p);
}
```

### Risk Assessment

Providing invalid size arguments to memory allocation functions can lead to buffer overflows and the execution of arbitrary code with the permissions of the vulnerable process.

<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>MEM35-C</td>
<td>High</td>
<td>Probable</td>
<td>High</td>
<td>P6</td>
<td>L2</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>Astrée</td>
<td>19.04</td>
<td>malloc-size-insufficient</td>
<td>Partially checked</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Besides direct rule violations, all undefined behaviour resulting from invalid memory accesses is reported by Astrée.</td>
</tr>
<tr>
<td>CodeSonar</td>
<td>5.2p0</td>
<td>ALLOC.SIZE. ADDOFLOW</td>
<td>Addition overflow of allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE:JOFLOW</td>
<td>Addition overflow of allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE. MULOFLOW</td>
<td>Multiplication overflow of allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE. SUBOFLOW</td>
<td>Subtraction underflow of allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ALLOC.SIZE.TRUNCIO</td>
<td>Truncation of allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>TAINT.SIZE</td>
<td>Tainted allocation size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MISC.MEM.SIZE.BAD</td>
<td>Unreasonable size argument</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td></td>
<td>BAD_ALLOC_STRLEN SIZECHECK (deprecated)</td>
<td>Could check violations of this rule by examining the size expression to malloc() or memcpy() functions. Specifically, the size argument should be bounded by 0, SIZE_MAX, and, unless it is a variable of type size_t or rsizen_t, it should be bounds-checked before the malloc() call. If the argument is of the expression a*b, then an appropriate check is if (a &lt; SIZE_MAX / b &amp;&amp; a &gt; 0) ...</td>
</tr>
<tr>
<td>Coverity</td>
<td>2017.07</td>
<td>BAD_ALLOC_STRLEN SIZECHECK</td>
<td>Partially implemented</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Can find instances where string length is miscalculated (length calculated may be one less than intended) for memory allocation purposes. Coverity Prevent cannot discover all violations of this rule, so further verification is necessary.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>INCORRECT. ALLOC_SIZE</td>
<td>Shows memory allocations that are assigned to a pointer that reference objects larger than the allocated block</td>
</tr>
<tr>
<td>Klocwork</td>
<td>2018</td>
<td>INCORRECT. ALLOC_SIZE</td>
<td>Enhanced enforcement</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>9.7.1</td>
<td>400 S, 487 S, 115 D</td>
<td>Enhanced enforcement</td>
</tr>
<tr>
<td>Splint</td>
<td>3.1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-MEM35-a</td>
<td>Do not use sizeof operator on pointer type to specify the size of the memory to be allocated via 'malloc', 'calloc' or 'realloc' function</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule MEM35-C</td>
<td>Checks for:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Pointer access out of bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Memory allocation with tainted size</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Rule fully covered</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>0696, 0701, 1069, 1071, 1073, 2840, 2841, 2842, 2843, 2935, 2936, 2937, 2938</td>
<td></td>
</tr>
<tr>
<td>PRQA QA-C++</td>
<td>4.4</td>
<td>2840, 2841, 2842, 2843, 2935, 2936, 2937, 2938</td>
<td></td>
</tr>
<tr>
<td>PVST-Studio</td>
<td>6.23</td>
<td>V531, V635, V641, V781</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>malloc-size-insufficient</td>
<td>Partially checked</td>
</tr>
<tr>
<td>TrustIn Soft Analyzer</td>
<td>1.38</td>
<td>mem_access</td>
<td>Exhaustively detects undefined behavior (see one compliant and one non-compliant example).</td>
</tr>
</tbody>
</table>

### Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://www.cert.org).”}

### 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>ARR01-C. Do not apply the sizeof operator to a pointer when taking the size of an array</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>INT31-C. Ensure that integer conversions do not result in lost or misinterpreted data</td>
<td></td>
</tr>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>INT32-C. Ensure that operations on signed integers do not result in overflow</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>INT18-C. Evaluate integer expressions in a larger size before comparing or assigning to that size</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>MEM04-C. Beware of zero-length allocations</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>ISO/IEC TR 24772:2013</td>
<td>Buffer Boundary Violation (Buffer Overflow) [HCB]</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>ISO/IEC TS 17961:2013</td>
<td>Taking the size of a pointer to determine the size of the pointed-to type [sizeofptr]</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-131, Incorrect Calculation of Buffer Size</td>
<td>2017-05-16: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-680</td>
<td>2017-05-18: CERT: Rule subset of CWE</td>
</tr>
<tr>
<td>CWE 2.11</td>
<td>CWE-789</td>
<td>2017-06-12: CERT: Partial overlap</td>
</tr>
</tbody>
</table>

#### CERT-CWE Mapping Notes

Key here for mapping notes

**CWE-680 and MEM35-C**

Intersection( INT32-C, MEM35-C) = Ø

CWE-680 = Union( MEM35-C, list) where list =

- Overflowed buffers with inadequate sizes not produced by integer overflow

**CWE-467 and MEM35-C**

CWE-467 = Subset( MEM35-C)

**CWE-789 and MEM35-C**

Intersection( MEM35-C, CWE-789) =

- Insufficient memory allocation on the heap

MEM35-C – CWE-789 =

- Insufficient memory allocation with trusted value but incorrect calculation

CWE-789 - MEM35-C =

- Sufficient memory allocation (possibly over-allocation) with untrusted value
CWE-120 and MEM35-C

Intersection( MEM35-C, CWE-120) = Ø

CWE-120 specifically addresses buffer overflow operations, which occur in the context of string-copying. MEM35-C specifically addresses allocation of memory ranges (some of which may be for subsequent string copy operations).

Consequently, they address different sections of code, although one (or both) may be responsible for a single buffer overflow vulnerability.

CWE-131 and MEM35-C

• Intersection( INT30-C, MEM35-C) = Ø

• CWE-131 = Union( MEM35-C, list) where list =

  • Miscalculating a buffer for a non-heap region (such as a variable-length array)

Bibliography

[Coverity 2007]
[Drepper 2006] Section 2.1.1, "Respecting Memory Bounds"
[Seacord 2013] Chapter 4, "Dynamic Memory Management"
               Chapter 5, "Integer Security"
[Viega 2005] Section 5.6.8, "Use of sizeof() on a Pointer Type"
[xorl 2009] CVE-2009-0587: Evolution Data Server Base64 Integer Overflows