

MEM04-C. Beware of zero-length allocations

When the requested size is 0, the behavior of the memory allocation functions `malloc()`, `calloc()`, and `realloc()` is [implementation-defined](#). Subclause 7.22.3 of the C Standard [ISO/IEC 9899:2011] states:

If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

In addition, the amount of storage allocated by a successful call to the allocation function when 0 bytes was requested is [unspecified](#). See [unspecified behavior 41](#) in subclause J.1 of the C Standard.

In cases where the memory allocation functions return a non-null pointer, reading from or writing to the allocated memory area results in [undefined behavior](#). Typically, the pointer refers to a zero-length block of memory consisting entirely of control structures. Overwriting these control structures damages the data structures used by the memory manager.

Noncompliant Code Example (`malloc()`)

The result of calling `malloc(0)` to allocate 0 bytes is implementation-defined. In this example, a dynamic array of integers is allocated to store `size` elements. However, if `size` is 0, the call to `malloc(size)` may return a reference to a block of memory of size 0 instead of a null pointer. When (nonempty) data is copied to this location, a heap-buffer overflow occurs.

```
size_t size;

/* Initialize size, possibly by user-controlled input */

int *list = (int *)malloc(size);
if (list == NULL) {
    /* Handle allocation error */
}
else {
    /* Continue processing list */
}
```

Compliant Solution (`malloc()`)

To ensure that 0 is never passed as a size argument to `malloc()`, `size` is checked to confirm it has a positive value:

```
size_t size;

/* Initialize size, possibly by user-controlled input */

if (size == 0) {
    /* Handle error */
}
int *list = (int *)malloc(size);
if (list == NULL) {
    /* Handle allocation error */
}
/* Continue processing list */
```

Noncompliant Code Example (`realloc()`)

The `realloc()` function deallocates the old object and returns a pointer to a new object of a specified size. If memory for the new object cannot be allocated, the `realloc()` function does not deallocate the old object, and its value is unchanged. If the `realloc()` function returns `NULL`, failing to free the original memory will result in a memory leak. As a result, the following idiom is often recommended for reallocating memory:

```

size_t nsize = /* Some value, possibly user supplied */;
char *p2;
char *p = (char *)malloc(100);
if (p == NULL) {
    /* Handle error */
}

/* ... */

if ((p2 = (char *)realloc(p, nsize)) == NULL) {
    free(p);
    p = NULL;
    return NULL;
}
p = p2;

```

However, this commonly recommended idiom has problems with zero-length allocations. If the value of `nsize` in this example is 0, the standard allows the option of either returning a null pointer or returning a pointer to an invalid (for example, zero-length) object. In cases where the `realloc()` function frees the memory but returns a null pointer, execution of the code results in a double-free vulnerability. If the `realloc()` function returns a non-null value, but the size was 0, the returned memory will be of size 0, and a heap overflow will occur if nonempty data is copied there.

Implementation Details

If this noncompliant code is compiled with GCC 3.4.6 and linked with libc 2.3.4, invoking `realloc(p, 0)` returns a non-null pointer to a zero-sized object (the same as `malloc(0)`). However, if the same code is compiled with either Microsoft Visual Studio or GCC 4.1.0, `realloc(p, 0)` returns a null pointer, resulting in a double-free vulnerability.

Compliant Solution (`realloc()`)

This compliant solution does not pass a size argument of zero to the `realloc()` function:

```

size_t nsize;
/* Initialize nsize */
char *p2;
char *p = (char *)malloc(100);
if (p == NULL) {
    /* Handle error */
}

/* ... */

p2 = NULL;
if (nsize != 0) {
    p2 = (char *)realloc(p, nsize);
}
if (p2 == NULL) {
    free(p);
    p = NULL;
    return NULL;
}
p = p2;

```

Risk Assessment

Allocating 0 bytes can lead to [abnormal program termination](#).

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM04-C	Low	Likely	Medium	P6	L2

Automated Detection

Tool	Version	Checker	Description
------	---------	---------	-------------

Astrée	19.04		Supported, but no explicit checker
CodeSonar	5.0p0	(customization)	Users can add a custom check for allocator calls with size argument 0 (this includes literal 0, unconstrained tainted values, and computed values).
Compass /ROSE			Can detect some violations of this rule. In particular, it warns when the argument to <code>malloc()</code> is a variable that has not been compared against 0 or that is known at compile time to be 0
Parasoft C /C++test	10.4.2	CERT_C-MEM04-a	The validity of values passed to library functions shall be checked
Polyspace Bug Finder	R2018a	Tainted sign change conversion Tainted size of variable length array Variable length array with nonpositive size	Value from an unsecure source changes sign Size of the variable-length array (VLA) is from an unsecure source and may be zero, negative, or too large Size of variable-length array is zero or negative

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](#).

Related Guidelines

SEI CERT C++ Coding Standard	VOID MEM04-CPP. Do not perform zero-length allocations
MITRE CWE	CWE-687 , Function call with incorrectly specified argument value

Bibliography

[ISO/IEC 9899:2011]	Section 7.22.3, "Memory Management Functions"
[Seacord 2013]	Chapter 4, "Dynamic Memory Management"
[Vanegue 2010]	" Automated Vulnerability Analysis of Zero-Sized Heap Allocations "

