SEI SERIES • A CERT® BOOK

Secure Coding in C and C++

Second Edition

Robert C. Seacord

Foreword by Richard D. Pethia
CERT Director

Secure Coding in C and C++

Second Edition

Initializing large blocks of memory can degrade performance and is not always necessary. The decision by the C standards committee to not require `malloc()` to initialize this memory reserves this decision for the programmer. If required, you can initialize memory using `memset()` or by calling `calloc()`, which zeros the memory. When calling `calloc()`, ensure that the arguments, when multiplied, do not wrap. *The CERT C Secure Coding Standard* [Seacord 2008], "MEM07-C. Ensure that the arguments to `calloc()`, when multiplied, can be represented as a `size_t`," further describes this problem.

Failing to initialize memory when required can also create a confidentiality or privacy risk. An example of this risk is the Sun tarball vulnerability [Graff 2003]. The tar program[3] is used to create archival files on UNIX systems. In this case, the tar program on Solaris 2.0 systems inexplicably included fragments of the `/etc/passwd` file, an example of an information leak that could compromise system security.

The problem in this case was that the `tar` utility failed to initialize the dynamically allocated memory it was using to read a block of data from the disk. Unfortunately, before allocating this block, the `tar` utility invoked a system call to look up user information from the `/etc/passwd` file. This memory chunk was deallocated by `free()` and then reallocated to the `tar` utility as the read buffer. The `free()` function is similar to `malloc()` in that neither is required to clear memory, and it would be unusual to find an implementation that did so. Sun fixed the Sun tarball vulnerability by replacing the call to `malloc()` with a call to `calloc()` in the `tar` utility. The existing solution is extremely fragile because any changes may result in the sensitive information being reallocated elsewhere in the program and leaked again, resulting in a *déjà vul* (a vulnerability that has "already been seen").

In cases like the Sun tarball vulnerability, where sensitive information is used, it is important to clear or overwrite the sensitive information before calling `free()`, as recommended by MEM03-C of *The CERT C Secure Coding Standard* [Seacord 2008]: "Clear sensitive information stored in reusable resources." Clearing or overwriting memory is typically accomplished by calling the C Standard `memset()` function. Unfortunately, compiler optimizations may silently remove a call to `memset()` if the memory is not accessed following the write. To avoid this possibility, you can use the `memset_s()` function defined in Annex K of the C Standard (if available). Unlike `memset()`, the `memset_s()` function assumes that the memory being set may be accessed in the future, and consequently the function call cannot be optimized away.

3. The UNIX tar (tape archive) command was originally designed to copy blocks of disk storage to magnetic tape. Today, tar is the predominant method of grouping files for transfer between UNIX systems.

See *The CERT C Secure Coding Standard* [Seacord 2008], "MSC06-C. Be aware of compiler optimization when dealing with sensitive data," for more information.

Failing to Check Return Values

Memory is a limited resource and can be exhausted. Available memory is typically bounded by the sum of the amount of physical memory and the swap space allocated to the operating system by the administrator. For example, a system with 1GB of physical memory configured with 2GB of swap space may be able to allocate, at most, 3GB of heap space to all running processes (minus the size of the operating system itself and the text and data segments of all running processes). Once all virtual memory is allocated, requests for more memory will fail. AIX and Linux have (nonconforming) behavior whereby allocation requests can succeed for blocks in excess of this maximum, but the kernel kills the process when it tries to access memory that cannot be backed by RAM or swap [Rodrigues 2009].

Heap exhaustion can result from a number of causes, including

- A memory leak (dynamically allocated memory is not freed after it is no longer needed; see the upcoming section "Memory Leaks")
- Incorrect implementation of common data structures (for example, hash tables or vectors)
- Overall system memory being exhausted, possibly because of other processes
- Transient conditions brought about by other processes' use of memory

The CERT C Secure Coding Standard [Seacord 2008], "MEM11-C. Do not assume infinite heap space," warns against memory exhaustion.

The return values for memory allocation functions indicate the failure or success of the allocation. The `aligned_alloc()`, `calloc()`, `malloc()`, and `realloc()` functions return null pointers if the requested memory allocation fails.

The application programmer must determine when an error has occurred and handle the error in an appropriate manner. Consequently, *The CERT C Secure Coding Standard* [Seacord 2008], "MEM32-C. Detect and handle memory allocation errors," requires that these errors be detected and properly managed.

C memory allocation functions return a null pointer if the requested space cannot be allocated. Example 4.2 shows a function that allocates memory using `malloc()` and tests the return value.

Example 4.2 Checking Return Codes from `malloc()`

```
01  int *create_int_array(size_t nelements_wanted) {
02    int *i_ptr = (int *)malloc(sizeof(int) * nelements_wanted);
03    if (i_ptr != NULL) {
04      memset(i_ptr, 0, sizeof(int) * nelements_wanted);
05    }
06    else {
07      return NULL;
08    }
09    return i_ptr;
10  }
```

When memory cannot be allocated, it is a good idea to have a consistent recovery plan, even if your solution is to print an error message and exit with a nonzero exit status.

Failure to detect and properly handle memory allocation errors can lead to unpredictable and unintended program behavior. For example, versions of Adobe Flash prior to 9.0.124.0 neglected to check the return value from `calloc()`, resulting in a vulnerability (VU#159523). Even when `calloc()` returns a null pointer, Flash writes to an offset from the return value. Dereferencing a null pointer usually results in a program crash, but dereferencing an offset from a null pointer allows an exploit to succeed without crashing the program.

"MEM32-C. Detect and handle memory allocation errors," in *The CERT C Secure Coding Standard* [Seacord 2008], contains another example of this problem. Assuming that `temp_num`, `tmp2`, and `num_of_records` are under the control of a malicious user in the following example, the attacker can cause `malloc()` to fail by providing a large value for `num_of_records`:

```
1  signal_info * start = malloc(num_of_records * sizeof(signal_info));
2  signal_info * point = (signal_info *)start;
3  point = start + temp_num - 1;
4  memcpy(point->sig_desc, tmp2, strlen(tmp2));
5  /* ... */
```

When `malloc()` fails, it returns a null pointer that is assigned to `start`. The value of `temp_num` is scaled by the size of `signal_info` when added to `start`. The resulting pointer value is stored in `point`. To exploit this vulnerability, the attacker can supply a value for `temp_num` that results in `point` referencing a writable address to which control is eventually transferred. The memory at that address is overwritten by the contents of the string referenced by `tmp2`, resulting in an arbitrary code execution vulnerability.

This vulnerability can be eliminated by simply testing that the pointer returned by `malloc()` is not null and handling the allocation error appropriately:

```
01  signal_info *start = malloc(num_of_records * sizeof(signal_info));
02  if (start == NULL) {
03    /* handle allocation error */
04  }
05  else {
06    signal_info *point = (signal_info *)start;
07    point = start + temp_num - 1;
08    memcpy(point->sig_desc, tmp2, strlen(tmp2));
09    /* ... */
10  }
```

Dereferencing Null or Invalid Pointers

The unary * operator denotes indirection. If the operand doesn't point to an object or function, the behavior of the unary * operator is undefined.

Among the invalid values for dereferencing a pointer by the unary * operator are a null pointer, an address inappropriately aligned for the type of object pointed to, and the address of an object after the end of its lifetime.

Dereferencing a null pointer typically results in a segmentation fault, but this is not always the case. For example, many Cray supercomputers had memory mapped at address 0, so it worked just like any other memory reference. Many embedded systems work the same way. Other embedded systems have registers mapped at address 0, so overwriting them can have unpredictable consequences. Each implementation is free to choose whatever works best for its environment, including considerations of performance, address space conservation, and anything else that might be relevant to the hardware or the implementation as a whole. In some situations, however, dereferencing a null pointer can lead to the execution of arbitrary code. *The CERT C Secure Coding Standard* [Seacord 2008], "EXP34-C. Do not dereference null pointers," further describes the problem of dereferencing a null pointer.

A real-world example of an exploitable null pointer dereference resulted from a vulnerable version of the `libpng` library as deployed on a popular ARM-based cell phone [Jack 2007]. The `libpng` library implements its own wrapper to `malloc()` that returns a null pointer on error or on being passed a 0-byte-length argument.

```
png_charp chunkdata;
chunkdata = (png_charp)png_malloc(png_ptr, length + 1);
```

The chunkdata pointer is later used as a destination argument in a call to memcpy(). Directly writing to a pointer returned from a memory allocation function is more common, but normally less exploitable, than using a pointer as an operand in pointer arithmetic.

If a length field of –1 is supplied to the code in this example, the addition wraps around to 0, and png_malloc() subsequently returns a null pointer, which is assigned to chunkdata. The subsequent call to memcpy() results in user-defined data overwriting memory starting at address 0. A write from or read to the memory address 0 will generally reference invalid or unused memory. In the case of the ARM and XScale architectures, the address 0 is mapped in memory and serves as the exception vector table.

Again, this vulnerability can be easily eliminated by ensuring that the pointer returned by malloc() or other memory allocation function or wrapper is not a null pointer. *The CERT C Secure Coding Standard* [Seacord 2008] rule violated in the example is "MEM35-C. Allocate sufficient memory for an object." The recommendation "MEM04-C. Do not perform zero-length allocations" is also violated.

Referencing Freed Memory

It is possible to access freed memory unless all pointers to that memory have been set to NULL or otherwise overwritten. (Unfortunately, the free() function cannot set its pointer argument to NULL because it takes a single argument of void * type and not void **.) An example of this programming error can be seen in the following loop [Kernighan 1988], which dereferences p after having first freed the memory referenced by p:

```
for (p = head; p != NULL; p = p->next)
  free(p);
```

The correct way to perform this operation is to save the required pointer before freeing:

```
1  for (p = head; p != NULL; p = q) {
2    q = p->next;
3    free(p);
4  }
```

Reading from freed memory is undefined behavior but almost always succeeds without a memory fault because freed memory is recycled by the memory manager. However, there is no guarantee that the contents of the memory have not been altered. Although the memory is usually not erased by a call

to free(), memory managers may use some of the space to manage free or *unallocated* memory. If the memory chunk has been reallocated, the entire contents may have been replaced. As a result, these errors may go undetected because the contents of memory may be preserved during testing but modified during operation.

Writing to a memory location that has been freed is also unlikely to result in a memory fault but could result in a number of serious problems. If the memory has been reallocated, a programmer may overwrite memory, believing that a memory chunk is *dedicated* to a particular variable when in reality it is being *shared*. In this case, the variable contains whatever data was written last. If the memory has not been reallocated, writing to a free chunk may overwrite and corrupt the data structures used by the memory manager. This can be (and has been) used as the basis for an exploit when the data being written is controlled by an attacker, as detailed later in this chapter.

Freeing Memory Multiple Times

Another dangerous error in managing dynamic memory is to free the same memory chunk more than once (the most common scenario being a *double-free*). This error is dangerous because it can corrupt the data structures in the memory manager in a manner that is not immediately apparent. This problem is exacerbated because many programmers do not realize that freeing the same memory multiple times can result in an exploitable vulnerability.

The sample program in Example 4.3 twice frees the memory chunk referenced by x: once on line 3 and again on line 6. This example is typical of a cut-and-paste error whereby a programmer cuts and pastes a block of code and then changes some element of it (often a variable name). In this example, it is easy to imagine that a programmer neglected to change the reference to x on line 6 into a reference to y, inadvertently freeing the memory twice (and leaking memory as well).

Example 4.3 Memory Referenced by x Freed Twice

```
1  x = malloc(n * sizeof(int));
2  /* access memory referenced by x */
3  free(x);
4  y = malloc(n * sizeof(int));
5  /* access memory referenced by y */
6  free(x);
```

The error may be less obvious if the elided statements to "access memory referenced by" x and y consist of many lines of code.