MEM00-C. Allocate and free memory in the same module, at the same level of abstraction

Dynamic memory management is a common source of programming flaws that can lead to security vulnerabilities. Poor memory management can lead to security issues, such as heap-buffer overflows, dangling pointers, and double-free issues [Seacord 2013]. From the programmer's perspective, memory management involves allocating memory, reading and writing to memory, and deallocating memory.

Allocating and freeing memory in different modules and levels of abstraction may make it difficult to determine when and if a block of memory has been freed, leading to programming defects, such as memory leaks, double-free vulnerabilities, accessing freed memory, or writing to freed or unallocated memory.

To avoid these situations, memory should be allocated and freed at the same level of abstraction and, ideally, in the same code module. This includes the use of the following memory allocation and deallocation functions described in subclause 7.23.3 of the C Standard [ISO/IEC 9899:2011]:

```
void *malloc(size_t size);

void *calloc(size_t nmemb, size_t size);

void *realloc(void *ptr, size_t size);

void *aligned_alloc(size_t alignment, size_t size);

void free(void *ptr);
```

Failing to follow this recommendation has led to real-world vulnerabilities. For example, freeing memory in different modules resulted in a vulnerability in MIT Kerberos 5 [MIT 2004]. The MIT Kerberos 5 code in this case contained error-handling logic, which freed memory allocated by the ASN.1 decoders if pointers to the allocated memory were non-null. However, if a detectable error occurred, the ASN.1 decoders freed the memory that they had allocated. When some library functions received errors from the ASN.1 decoders, they also attempted to free the same memory, resulting in a double-free vulnerability.

Noncompliant Code Example

This noncompliant code example shows a double-free vulnerability resulting from memory being allocated and freed at differing levels of abstraction. In this example, memory for the `list` array is allocated in the `process_list()` function. The array is then passed to the `verify_size()` function that performs error checking on the size of the list. If the size of the list is below a minimum size, the memory allocated to the list is freed, and the function returns to the caller. The calling function then frees this same memory again, resulting in a double-free and potentially exploitable vulnerability.

```
enum { MIN_SIZE_ALLOWED = 32 };

int verify_size(char *list, size_t size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle error condition */
    free(list);
    return -1;
  }
  return 0;
}

void process_list(size_t number) {
  char *list = (char *)malloc(number);
  if (list == NULL) {
    /* Handle allocation error */
  }

  if (verify_size(list, number) == -1) {
      free(list);
      return;
  }

  /* Continue processing list */

  free(list);
}
```

The call to free memory in the `verify_size()` function takes place in a subroutine of the `process_list()` function, at a different level of abstraction from the allocation, resulting in a violation of this recommendation. The memory deallocation also occurs in error-handling code, which is frequently not as well tested as "green paths" through the code.

Compliant Solution

To correct this problem, the error-handling code in `verify_size()` is modified so that it no longer frees `list`. This change ensures that `list` is freed only once, at the same level of abstraction, in the `process_list()` function.

```
enum { MIN_SIZE_ALLOWED = 32 };

int verify_size(const char *list, size_t size) {
  if (size < MIN_SIZE_ALLOWED) {
    /* Handle error condition */
    return -1;
  }
  return 0;
}

void process_list(size_t number) {
  char *list = (char *)malloc(number);

  if (list == NULL) {
    /* Handle allocation error */
  }

  if (verify_size(list, number) == -1) {
      free(list);
      return;
  }

  /* Continue processing list */

  free(list);
}
```

Risk Assessment

The mismanagement of memory can lead to freeing memory multiple times or writing to already freed memory. Both of these coding errors can result in an attacker executing arbitrary code with the permissions of the vulnerable process. Memory management errors can also lead to resource depletion and denial-of-service attacks.

Recommendation	Severity	Likelihood	Remediation Cost	Priority	Level
MEM00-C	High	Probable	Medium	P12	L1

Automated Detection

Tool	Version	Checker	Description
CodeSonar	5.2p0	**ALLOC.DF** **ALLOC.LEAK**	Double free Leak
Compass/ROSE			Could detect possible violations by reporting any function that has `malloc()` or `free()` but not both. This would catch some false positives, as there would be no way to tell if `malloc()` and `free()` are at the same level of abstraction if they are in different functions
Coverity	6.5	**RESOURCE_LEAK**	Fully implemented

Klocwork	2018	**FREE.INCONSISTENT** **UFM.FFM.MIGHT** **UFM.FFM.MUST** **UFM.DEREF.MIGHT** **UFM.DEREF.MUST** **UFM.RETURN.MIGHT** **UFM.RETURN.MUST** **UFM.USE.MIGHT** **UFM.USE.MUST** **MLK.MIGHT** **MLK.MUST** **MLK.RET.MIGHT** **MLK.RET.MUST** **FNH.MIGHT** **FNH.MUST** **FUM.GEN.MIGHT** **FUM.GEN.MUST** **RH.LEAK**	
LDRA tool suite	9.7.1	**50 D**	Partially implemented
Parasoft C/C++test	10.4.2	**CERT_C-MEM00-a** **CERT_C-MEM00-b** **CERT_C-MEM00-c** **CERT_C-MEM00-d** **CERT_C-MEM00-e**	Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not allocate memory and expect that someone else will deallocate it later Do not use resources that have been freed Ensure resources are freed
Parasoft Insure++			Runtime analysis
Polyspace Bug Finder	R2019b	CERT C: Rec. MEM00-C	Checks for: • Invalid free of pointer • Deallocation of previously deallocated pointer • Use of previously freed pointer Rec. partially covered.

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	VOID MEM11-CPP. Allocate and free memory in the same module, at the same level of abstraction
ISO/IEC TR 24772:2013	Memory Leak [XYL]
MITRE CWE	CWE-415, Double free CWE-416, Use after free

Bibliography

[MIT 2004]	
[Plakosh 2005]	
[Seacord 2013]	Chapter 4, "Dynamic Memory Management"