MEM11-C. Do not assume infinite heap space

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 total 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 has been allocated, requests for more memory will fail. As discussed in ERR33-C. Detect and handle standard library errors, programs that fail to check for and properly handle memory allocation failures will have undefined behavior and are likely to crash when heap space is exhausted. Heap exhaustion can result from

- A memory leak
- An infinite loop
- The program requiring more memory than is present by default in the heap
- Incorrect implementation of common data structures (for example, hash tables or vectors)
- Overall system memory being exhausted, possibly because of other processes
- The maximum size of a process's data segment being exceeded

If malloc() is unable to return the requested memory, it returns NULL instead.

However, simply checking for and handling memory allocation failures may not be sufficient. Programs such as long-servers that manipulate large data sets need to be designed in a way that permits them to deliver their services when system resources, including the heap, are in short supply. Making use of additional storage devices, such as disk space or databases, is essential in such systems.

Noncompliant Code Example

This example places no upper bound on the memory space required. As a result, the program can easily exhaust the heap. A heap error will be generated if the heap continues to be populated, even if there is no space available.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

enum {MAX_LENGTH=100};

typedef struct namelist_s {
    char name[MAX_LENGTH];
    struct namelist_s* next;
} *namelist_t;

int main() {
    namelist_t names = NULL;
    char new_name[MAX_LENGTH];

    do {
        /*
         * Adding unknown number of records to a list;
         * the user can enter as much data as he wants
         * and exhaust the heap.
         */
        puts("To quit, enter \"quit\"");
        puts("Enter record:");
        fgets(new_name, MAX_LENGTH, stdin);
        if (strcmp(new_name, "quit") != 0) {
            /* Names continue to be added without bothering
             * about the size on the heap.
             */
            unsigned int i = strlen(new_name) - 1;
            if (new_name[i] == '\n') new_name[i] = '\0';
            namelist_t new_entry = (namelist_t) malloc(sizeof(struct namelist_s));
            if (new_entry == NULL) {
                /* Handle error */
            }
            strcpy( new_entry->name, new_name);
            new_entry->next = names;
            names = new_entry;
        }
        puts(new_name);
    } while (strcmp( new_name, "quit") != 0);

    return 0;
}

Compliant Solution

If the objects or data structures are large enough to potentially cause heap exhaustion, the programmer should consider using databases instead to ensure that records are written to the disk and that the data structure does not outgrow the heap.

In the previous noncompliant code example, the user can reuse a single long variable to store the input and write that value into a simple database containing a table User with a field userID along with any other required fields. This prevents the heap from being exhausted.

Risk Assessment

It is difficult to pinpoint violations of this recommendation because static analysis tools are currently unable to identify code that can lead to heap exhaustion. The heap size also varies for different runtime environments.

Automated Detection
Related Vulnerabilities

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

Related Guidelines

<table>
<thead>
<tr>
<th>SEI CERT C++ Coding Standard</th>
<th>VOID MEM12-CPP. Do not assume infinite heap space</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT Oracle Secure Coding Standard for Java</td>
<td>MSC05-J. Do not exhaust heap space</td>
</tr>
<tr>
<td>MITRE CWE</td>
<td>CWE-770, Allocation of resources without limits or throttling</td>
</tr>
</tbody>
</table>