MSC19-C. For functions that return an array, prefer returning an empty array over a null value

Many functions have the option of returning a pointer to an object or returning NULL if a valid pointer cannot be produced. Some functions return arrays, which appear like a pointer to an object. However, if a function has the option of returning an array or indicating that a valid array is not possible, it should not return NULL. Instead, the function should return an empty array. Often, code that calls a function that returns an array intends merely to iterate over the array elements. In this case, the calling code need not change—iterating over the elements works correctly even if the returned array is empty, so the calling code need not check the return value for NULL.

This situation is complicated by the fact that C does not keep track of the length of an array. However, two popular methods have emerged to emulate this behavior. The first is to wrap the array in a struct with an integer storing the length. The second is to place a sentinel value at the end of the data in the array. This second approach is most commonly manifested in null-terminated byte strings (NTBSs).

Noncompliant Code Example (Struct)

In this noncompliant code example, an inventory system keeps track of the total number of different items (denoted length). Each item is given an index in the array, and the value for that index is the stock of that item. Adding a new item increases length in the struct. Stocking more of an item increases the value for that item's index. For example, if 5 books and 2 erasers are in stock, the inventory would be stockOfItem[0] = 5 and stockOfItem[1] = 2, assuming books are index 0 and erasers are index 1.

The problem arises in this setup when no items are being stocked. getStock would recognize that length = 0 and would return NULL. In this noncompliant code example, erroneous behavior results from getStock returning NULL while main neglects to check for such a value. It results in an abnormal program termination after returning to the main function.

```c
#include <stdio.h>
enum { INV_SIZE=20 };

typedef struct {
    size_t stockOfItem[INV_SIZE];
    size_t length;
} Inventory;

size_t *getStock(Inventory iv);

int main(void) {
    Inventory iv;
    size_t *item;

    iv.length = 0;

    /*
     * Other code that might modify the inventory but still
     * leave no items in it upon completion.
     */
    item = getStock(iv);

    printf("Stock of first item in inventory: \%zd\n", item[0]);

    return 0;
}

size_t *getStock(Inventory iv) {
    if (iv.length == 0) {
        return NULL;
    } else {
        return iv.stockOfItem;
    }
}
```

Compliant Solution

This compliant solution eliminates the NULL return and returns the item array, even if it is zero-length. The main function can effectively handle this situation without exhibiting erroneous behavior. Since the array lives on the stack, it must prevent returning a value in the stack frame (as mandated by DC L30-C. Declare objects with appropriate storage durations). So the getStack() function also takes a pointer to Inventory, so that it can return a pointer to its contents safely.
#include <stdio.h>

enum { INV_SIZE=20 }

typedef struct {
  size_t stockOfItem[INV_SIZE];
  size_t length;
} Inventory;

size_t *getStock(Inventory* iv);

int main(void) {
  Inventory iv;
  size_t *item;

  iv.length = 0;
  
  /*
   * Other code that might modify the inventory but still
   * leave no items in it upon completion.
   */
  item = getStock(&iv);

  if (iv.length != 0) {
    printf("Stock of first item in inventory: %zd\n", item[0]);
  }

  return 0;
}

size_t *getStock(Inventory* iv) {
  return iv->stockOfItem;
}

Noncompliant Code Example (Sentinel Value)

This noncompliant code example implements an inventory system similar to the one described previously. However, instead of storing the length of the array in a struct, a sentinel value of FINAL_ITEM is used. The value for the index following the last item is set as FINAL_ITEM. It is assumed that out-of-stock items (assigned value 0) are removed from the array, and the contents of later items are shifted to lower indexes.

The following code attempts to return an array of the items in stock, sorted by the amount of each item in stock. The arraySort function incorrectly returns NULL instead of a pointer to an empty array when no items are in stock. The null return is improperly handled by the main function, which is attempting to print out the returned array, and an abnormal program termination results.

#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>

enum { FINAL_ITEM=SIZE_MAX, INV_SIZE=20 };  

size_t *arraySort(size_t *array);

int main(void) {
    size_t i;
    size_t stockOfItem[INV_SIZE];
    size_t *sortedArray;

    /* Other code that might use stockarray but leaves it empty */

    sortedArray = arraySort(stockOfItem);

    for (i = 0; sortedArray[i] != FINAL_ITEM; i++) {
        printf("Item stock: %zd, stockOfItem[%zd]");
    }

    return 0;
}

/* Create new sorted array */
size_t *arraySort(size_t *array) {
    size_t i;
    size_t *sortedArray;

    for (i = 0; array[i] != FINAL_ITEM; i++) {
        if (i == 0) {
            return NULL;
        }
    }

    sortedArray = (size_t*) malloc(sizeof(size_t)*i);
    if (sortedArray == NULL) {
        /* Handle memory error */
    }

    /* Add sorted data to array */
    return sortedArray;
}
#include <stdio.h>
#include <stdint.h>
#include <malloc.h>

enum { FINAL_ITEM=SIZE_MAX, INV_SIZE=20 };  

size_t *arraySort(size_t *array);  

int main(void) {    
    size_t i;    
    size_t stockOfItem[INV_SIZE];    
    size_t *sortedArray;  

    /* Other code that might use stockarray but leaves it empty */  
    sortedArray = arraySort(stockOfItem);  

    for (i = 0; sortedArray[i] != FINAL_ITEM; i++) {  
        printf("Item stock: %zd", sortedArray[i]);  
    }  

    return 0;  
}  

/* Create new sorted array */  
size_t *arraySort(size_t *array) {    
    size_t i;    
    size_t *sortedArray;  

    for(i = 0; array[i] != FINAL_ITEM; i++);  

    if (i == 0) {  
        size_t *emptyArray = (size_t*) malloc(sizeof(size_t));  
        if (emptyArray == NULL) {  
            /* Handle memory error */  
        }  
        emptyArray[0] = FINAL_ITEM;  
        return emptyArray;  
    }  

    sortedArray = (size_t*) malloc(sizeof(size_t)*i);  
    if (sortedArray == NULL) {  
        /* Handle memory error */  
    }  

    /* Add sorted data to array */  

    return sortedArray;  
}

Risk Assessment

Returning NULL rather than a zero-length array can lead to vulnerabilities when the client code does not handle NULL properly. Abnormal program termination can result when the calling function performs operations on NULL.

<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>MSC19-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>High</td>
<td>P1</td>
<td>L3</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>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-MSC19-a</td>
<td>Avoid accessing arrays out of bounds</td>
</tr>
<tr>
<td></td>
<td></td>
<td>CERT_C-MSC19-b</td>
<td>Avoid null pointer dereferencing</td>
</tr>
</tbody>
</table>
Related Vulnerabilities

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

Bibliography

[Bloch 2008] Item 43, "Return Empty Arrays or Collections, Not Nulls"