MEM33-C. Allocate and copy structures containing a flexible array member dynamically

The C Standard, 6.7.2.1, paragraph 18 [ISO/IEC 9899:2011], says

As a special case, the last element of a structure with more than one named member may have an incomplete array type; this is called a flexible array member. In most situations, the flexible array member is ignored. In particular, the size of the structure is as if the flexible array member were omitted except that it may have more trailing padding than the omission would imply.

The following is an example of a structure that contains a flexible array member:

```c
struct flex_array_struct {
    int num;
    int data[];
};
```

This definition means that when computing the size of such a structure, only the first member, `num`, is considered. Unless the appropriate size of the flexible array member has been explicitly added when allocating storage for an object of the `struct` the result of accessing the member `data` of a variable of nonpointer type `struct flex_array_struct` is undefined. DCL38-C. Use the correct syntax when declaring a flexible array member describes the correct way to declare a `struct` with a flexible array member.

To avoid the potential for undefined behavior, structures that contain a flexible array member should always be allocated dynamically. Flexible array structures must

- Have dynamic storage duration (be allocated via `malloc()` or another dynamic allocation function)
- Be dynamically copied using `memcpy()` or a similar function and not by assignment
- When used as an argument to a function, be passed by pointer and not copied by value

Noncompliant Code Example (Storage Duration)

This noncompliant code example uses automatic storage for a structure containing a flexible array member:

```c
#include <stddef.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(void) {
    struct flex_array_struct flex_struct;
    size_t array_size = 4;
    /* Initialize structure */
    flex_struct.num = array_size;
    for (size_t i = 0; i < array_size; ++i) {
        flex_struct.data[i] = 0;
    }
}
```

Because the memory for `flex_struct` is reserved on the stack, no space is reserved for the `data` member. Accessing the `data` member is undefined behavior.

Compliant Solution (Storage Duration)

This compliant solution dynamically allocates storage for `flex_array_struct`:
Noncompliant Code Example (Copying)

This noncompliant code example attempts to copy an instance of a structure containing a flexible array member (struct flex_array_struct) by assignment:

```c
#include <stddef.h>
struct flex_array_struct {
    size_t num;
    int data[];
};
void func(struct flex_array_struct *struct_a, struct flex_array_struct *struct_b) {
    *struct_b = *struct_a;
}
```

When the structure is copied, the size of the flexible array member is not considered, and only the first member of the structure, num, is copied, leaving the array contents untouched.

Compliant Solution (Copying)

This compliant solution uses memcpy() to properly copy the content of struct_a into struct_b:

```c
#include <stdlib.h>
struct flex_array_struct {
    size_t num;
    int data[];
};
void func(void) {
    struct flex_array_struct *flex_struct;
    size_t array_size = 4;
    flex_struct = (struct flex_array_struct *)malloc(sizeof(struct flex_array_struct) + sizeof(int) * array_size);
    if (flex_struct == NULL) {
        /* Handle error */
    }
    /* Initialize structure */
    flex_struct->num = array_size;
    for (size_t i = 0; i < array_size; ++i) {
        flex_struct->data[i] = 0;
    }
}
```
```c
#include <string.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void func(struct flex_array_struct *struct_a,
          struct flex_array_struct *struct_b) {
    if (struct_a->num > struct_b->num) {
        /* Insufficient space; handle error */
        return;
    }
    memcpy(struct_b, struct_a,
           sizeof(struct flex_array_struct) + (sizeof(int)
               * struct_a->num));
}
```

**Noncompliant Code Example (Function Arguments)**

In this noncompliant code example, the flexible array structure is passed by value to a function that prints the array elements:

```c
#include <stdio.h>
#include <stdlib.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void print_array(struct flex_array_struct struct_p) {
    puts("Array is: ");
    for (size_t i = 0; i < struct_p.num; ++i) {
        printf("%d ", struct_p.data[i]);
    }
    putchar('
');
}

void func(void) {
    struct flex_array_struct *struct_p;
    size_t array_size = 4;

    /* Space is allocated for the struct */
    struct_p = (struct flex_array_struct *)malloc(
                sizeof(struct flex_array_struct)
                + sizeof(int) * array_size);
    if (struct_p == NULL) {
        /* Handle error */
    }
    struct_p->num = array_size;
    for (size_t i = 0; i < array_size; ++i) {
        struct_p->data[i] = i;
    }
    print_array(*struct_p);
}
```

Because the argument is passed by value, the size of the flexible array member is not considered when the structure is copied, and only the first member of the structure, `num`, is copied.

**Compliant Solution (Function Arguments)**

In this compliant solution, the structure is passed by reference and not by value:

```c
```
```c
#include <stdio.h>
#include <stdlib.h>

struct flex_array_struct {
    size_t num;
    int data[];
};

void print_array(struct flex_array_struct *struct_p) {
    puts("Array is: ");
    for (size_t i = 0; i < struct_p->num; ++i) {
        printf("%d ", struct_p->data[i]);
    }
    putchar('n');
}

void func(void) {
    struct flex_array_struct *struct_p;
    size_t array_size = 4;
    /* Space is allocated for the struct and initialized... */
    print_array(struct_p);
}
```

## Risk Assessment

Failure to use structures with flexible array members correctly can result in **undefined behavior**.

<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>MEM33-C</td>
<td>Low</td>
<td>Unlikely</td>
<td>Low</td>
<td>P3</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>Astrée</td>
<td>19.04</td>
<td>flexible-array-member-assignment flexible-array-member-declaration</td>
<td>Fully checked</td>
</tr>
<tr>
<td>Axivion Bauhaus Suite</td>
<td>6.9.0</td>
<td>CertC-MEM33</td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Compass/ROSE</td>
<td>9.7.1</td>
<td></td>
<td>Can detect all of these</td>
</tr>
<tr>
<td>LDRA tool suite</td>
<td>649 S, 650 S</td>
<td></td>
<td>Fully implemented</td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>10.4.2</td>
<td>CERT_C-MEM333-a CERT_C-MEM33-b</td>
<td>Allocate structures containing a flexible array member dynamically</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Do not copy instances of structures containing a flexible array member</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2019b</td>
<td>CERT C: Rule MEM33-C</td>
<td>Checks for misuse of structure with flexible array member (rule fully covered)</td>
</tr>
<tr>
<td>PRQA QA-C</td>
<td>9.7</td>
<td>1061, 1062, 1063, 1064</td>
<td></td>
</tr>
<tr>
<td>RuleChecker</td>
<td>19.04</td>
<td>flexible-array-member-assignment flexible-array-member-declaration</td>
<td>Fully checked</td>
</tr>
</tbody>
</table>

## Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the [CERT website](https://www.cse.columbia.edu/cert/).

## Related Guidelines

**Key here** (explains table format and definitions)

<table>
<thead>
<tr>
<th>Taxonomy</th>
<th>Taxonomy item</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>CERT C Secure Coding Standard</td>
<td>DCL38-C. Use the correct syntax when declaring a flexible array member</td>
<td>Prior to 2018-01-12: CERT: Unspecified Relationship</td>
</tr>
</tbody>
</table>
CERT-CWE Mapping Notes

Key here for mapping notes

CWE-401 and MEM33-CPP

There is no longer a C++ rule for MEM33-CPP. (In fact, all C++ rules from 30-50 are gone, because we changed the numbering system to be 50-99 for C++ rules.)

Bibliography

<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO/IEC 9899:2011</td>
<td>Subclause 6.7.2.1, &quot;Structure and Union Specifiers&quot;</td>
</tr>
<tr>
<td>JTC1/SC22/WG14 N791</td>
<td>Solving the Struct Hack Problem</td>
</tr>
</tbody>
</table>