ARR32-C. Ensure size arguments for variable length arrays are in a valid range

Variable length arrays (VLAs), a conditionally supported language feature, are essentially the same as traditional C arrays except that they are declared with a size that is not a constant integer expression and can be declared only at block scope or function prototype scope and no linkage. When supported, a variable length array can be declared

```
{ /* Block scope */
  char vla[size];
}
```

where the integer expression `size` and the declaration of `vla` are both evaluated at runtime. If the size argument supplied to a variable length array is not a positive integer value, the behavior is undefined. (See undefined behavior 75.) Additionally, if the magnitude of the argument is excessive, the program may behave in an unexpected way. An attacker may be able to leverage this behavior to overwrite critical program data [Griffiths 2006]. The programmer must ensure that size arguments to variable length arrays, especially those derived from untrusted data, are in a valid range.

Because variable length arrays are a conditionally supported feature of C11, their use in portable code should be guarded by testing the value of the macro `__STDC_NO_VLA__`. Implementations that do not support variable length arrays indicate it by setting `__STDC_NO_VLA__` to the integer constant 1.

Noncompliant Code Example

In this noncompliant code example, a variable length array of size `size` is declared. The `size` is declared as `size_t` in compliance with INT01-C. Use rsize_t or size_t for all integer values representing the size of an object.

```
#include <stddef.h>

extern void do_work(int *array, size_t size);

void func(size_t size) {
  int vla[size];
  do_work(vla, size);
}
```

However, the value of `size` may be zero or excessive, potentially giving rise to a security vulnerability.

Compliant Solution

This compliant solution ensures the `size` argument used to allocate `vla` is in a valid range (between 1 and a programmer-defined maximum); otherwise, it uses an algorithm that relies on dynamic memory allocation. The solution also avoids unsigned integer wrapping that, given a sufficiently large value of `size`, would cause `malloc` to allocate insufficient storage for the array.

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

enum { MAX_ARRAY = 1024 };
extern void do_work(int *array, size_t size);

void func(size_t size) {
  if (0 == size || SIZE_MAX / sizeof(int) < size) {
    /* Handle error */
    return;
  }
  if (size < MAX_ARRAY) {
    int vla[size];
    do_work(vla, size);
  } else {
    int *array = (int *)malloc(size * sizeof(int));
    if (array == NULL) {
      /* Handle error */
    }
    do_work(array, size);
    free(array);
  }
}
```

Noncompliant Code Example (sizeof)

The following noncompliant code example defines A to be a variable length array and then uses the sizeof operator to compute its size at runtime. When the function is called with an argument greater than SIZE_MAX / (N1 * sizeof (int)), the runtime sizeof expression may wrap around, yielding a result that is smaller than the mathematical product N1 * n2 * sizeof (int). The call to malloc(), when successful, will then allocate storage for fewer than n2 elements of the array, causing one or more of the final memset() calls in the for loop to write past the end of that storage.

```
#include <stdlib.h>
#include <string.h>

enum { N1 = 4096 };

void *func(size_t n2) {
  typedef int A[n2][N1];

  A *array = malloc(sizeof(A));
  if (!array) {
    /* Handle error */
    return NULL;
  }

  for (size_t i = 0; i != n2; ++i) {
    memset(array[i], 0, N1 * sizeof(int));
  }

  return array;
}
```

Compliant Solution (sizeof)

This compliant solution prevents sizeof wrapping by detecting the condition before it occurs and avoiding the subsequent computation when the condition is detected.

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

enum { N1 = 4096 };

void *func(size_t n2) {
  if (n2 > SIZE_MAX / (N1 * sizeof(int))) {
    /* Prevent sizeof wrapping */
    return NULL;
  }

  typedef int A[n2][N1];

  A *array = malloc(sizeof(A));
  if (!array) {
    /* Handle error */
    return NULL;
  }

  for (size_t i = 0; i != n2; ++i) {
    memset(array[i], 0, N1 * sizeof(int));
  }
  return array;
}
```

Implementation Details

Microsoft

Variable length arrays are not supported by Microsoft compilers.

Risk Assessment

Failure to properly specify the size of a variable length array may allow arbitrary code execution or result in stack exhaustion.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR32-C	High	Probable	High	**P6**	**L2**

Automated Detection

Tool	Version	Checker	Description
Coverity	2017.07	**REVERSE_NEGATIVE**	Fully implemented
LDRA tool suite	9.7.1	**621 S**	Enhanced enforcement
Parasoft C/C++test	10.4.2	**CERT_C-ARR32-a**	Ensure the size of the variable length array is in valid range
Polyspace Bug Finder	R2019a	CERT C: Rule ARR32-C	Checks for: • Memory allocation with tainted size • Tainted size of variable length array Rule fully covered.
PRQA QA-C	9.5	**1051, 2052**	Partially implemented
Cppcheck	1.66	**negativeArraySize**	Context sensitive analysis Will warn only if given size is negative
TrustInSoft Analyzer	1.38	**alloca_bounds**	Exhaustively verified.

Related Vulnerabilities

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

Related Guidelines

Key here (explains table format and definitions)

Taxonomy	Taxonomy item	Relationship
CERT C Secure Coding Standard	INT01-C. Use rsize_t or size_t for all integer values representing the size of an object	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Unchecked Array Indexing [XYZ]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961:2013	Tainted, potentially mutilated, or out-of-domain integer values are used in a restricted sink [taintsink]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-758	2017-06-29: CERT: Rule subset of CWE

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-129 and ARR32-C

Intersection(CWE-188, EXP39-C) = Ø

ARR32-C addresses specifying the size of a variable-length array (VLA). CWE-129 addresses invalid array indices, not array sizes.

CWE-758 and ARR32-C

Independent(INT34-C, INT36-C, MSC37-C, FLP32-C, EXP33-C, EXP30-C, ERR34-C, ARR32-C)

CWE-758 = Union(ARR32-C, list) where list =

- Undefined behavior that results from anything other than too large a VLA dimension.

CWE-119 and ARR32-C

- Intersection(CWE-119, ARR32-C) = Ø

- ARR32-C is not about providing a valid buffer but reading/writing outside it. It is about providing an invalid buffer, or one that exhausts the stack.

Bibliography

[Griffiths 2006]

← ↑ →