EXP33-C. Do not read uninitialized memory

Local, automatic variables assume unexpected values if they are read before they are initialized. The C Standard, 6.7.9, paragraph 10, specifies [ISO/IEC 9899:2011]

> *If an object that has automatic storage duration is not initialized explicitly, its value is indeterminate.*

See undefined behavior 11.

When local, automatic variables are stored on the program stack, for example, their values default to whichever values are currently stored in stack memory.

Additionally, some dynamic memory allocation functions do not initialize the contents of the memory they allocate.

Function	Initialization
`aligned_alloc()`	Does not perform initialization
`calloc()`	Zero-initializes allocated memory
`malloc()`	Does not perform initialization
`realloc()`	Copies contents from original pointer; may not initialize all memory

Uninitialized automatic variables or dynamically allocated memory has indeterminate values, which for objects of some types, can be a trap representation. Reading such trap representations is undefined behavior; it can cause a program to behave in an unexpected manner and provide an avenue for attack. (See undefined behavior 10 and undefined behavior 12.) In many cases, compilers issue a warning diagnostic message when reading uninitialized variables. (See MSC00-C. Compile cleanly at high warning levels for more information.)

Noncompliant Code Example (Return-by-Reference)

In this noncompliant code example, the `set_flag()` function is intended to set the parameter, `sign_flag`, to the sign of `number`. However, the programmer neglected to account for the case where `number` is equal to `0`. Because the local variable `sign` is uninitialized when calling `set_flag()` and is never written to by `set_flag()`, the comparison operation exhibits undefined behavior when reading `sign`.

```
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) {
    return;
  }

  if (number > 0) {
    *sign_flag = 1;
  } else if (number < 0) {
    *sign_flag = -1;
  }
}

int is_negative(int number) {
  int sign;
  set_flag(number, &sign);
  return sign < 0;
}
```

Some compilers assume that when the address of an uninitialized variable is passed to a function, the variable is initialized within that function. Because compilers frequently fail to diagnose any resulting failure to initialize the variable, the programmer must apply additional scrutiny to ensure the correctness of the code.

This defect results from a failure to consider all possible data states. (See MSC01-C. Strive for logical completeness for more information.)

Compliant Solution (Return-by-Reference)

This compliant solution trivially repairs the problem by accounting for the possibility that `number` can be equal to 0.

Although compilers and static analysis tools often detect uses of uninitialized variables when they have access to the source code, diagnosing the problem is difficult or impossible when either the initialization or the use takes place in object code for which the source code is inaccessible. Unless doing so is prohibitive for performance reasons, an additional defense-in-depth practice worth considering is to initialize local variables immediately after declaration.

```
void set_flag(int number, int *sign_flag) {
  if (NULL == sign_flag) {
    return;
  }

  /* Account for number being 0 */
  if (number >= 0) {
    *sign_flag = 1;
  } else {
    *sign_flag = -1;
  }
}

int is_negative(int number) {
  int sign = 0; /* Initialize for defense-in-depth */
  set_flag(number, &sign);
  return sign < 0;
}
```

Noncompliant Code Example (Uninitialized Local)

In this noncompliant code example, the programmer mistakenly fails to set the local variable `error_log` to the `msg` argument in the `report_error()` function [Mercy 2006]. Because `error_log` has not been initialized, an indeterminate value is read. The `sprintf()` call copies data from the arbitrary location pointed to by the indeterminate `error_log` variable until a null byte is reached, which can result in a buffer overflow.

```
#include <stdio.h>

/* Get username and password from user, return -1 on error */
extern int do_auth(void);
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
  const char *error_log;
  char buffer[BUFFERSIZE];

  sprintf(buffer, "Error: %s", error_log);
  printf("%s\n", buffer);
}

int main(void) {
  if (do_auth() == -1) {
    report_error("Unable to login");
  }
  return 0;
}
```

Noncompliant Code Example (Uninitialized Local)

In this noncompliant code example, the `report_error()` function has been modified so that `error_log` is properly initialized:

```
#include <stdio.h>
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
  const char *error_log = msg;
  char buffer[BUFFERSIZE];

  sprintf(buffer, "Error: %s", error_log);
  printf("%s\n", buffer);
}
```

This example remains problematic because a buffer overflow will occur if the null-terminated byte string referenced by `msg` is greater than 17 characters, including the null terminator. (See STR31-C. Guarantee that storage for strings has sufficient space for character data and the null terminator for more information.)

Compliant Solution (Uninitialized Local)

In this compliant solution, the buffer overflow is eliminated by calling the `snprintf()` function:

```
#include <stdio.h>
enum { BUFFERSIZE = 24 };
void report_error(const char *msg) {
  char buffer[BUFFERSIZE];

  if (0 < snprintf(buffer, BUFFERSIZE, "Error: %s", msg))
    printf("%s\n", buffer);
  else
    puts("Unknown error");
}
```

Compliant Solution (Uninitialized Local)

A less error-prone compliant solution is to simply print the error message directly instead of using an intermediate buffer:

```
#include <stdio.h>

void report_error(const char *msg) {
  printf("Error: %s\n", msg);
}
```

Noncompliant Code Example (`mbstate_t`)

In this noncompliant code example, the function `mbrlen()` is passed the address of an automatic `mbstate_t` object that has not been properly initialized. This is undefined behavior 200 because `mbrlen()` dereferences and reads its third argument.

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

void func(const char *mbs) {
  size_t len;
  mbstate_t state;

  len = mbrlen(mbs, strlen(mbs), &state);
}
```

Compliant Solution (`mbstate_t`)

Before being passed to a multibyte conversion function, an `mbstate_t` object must be either initialized to the initial conversion state or set to a value that corresponds to the most recent shift state by a prior call to a multibyte conversion function. This compliant solution sets the `mbstate_t` object to the initial conversion state by setting it to all zeros:

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

void func(const char *mbs) {
  size_t len;
  mbstate_t state;

  memset(&state, 0, sizeof(state));
  len = mbrlen(mbs, strlen(mbs), &state);
}
```

Noncompliant Code Example (POSIX, Entropy)

In this noncompliant code example described in "More Randomness or Less" [Wang 2012], the process ID, time of day, and uninitialized memory `junk` is used to seed a random number generator. This behavior is characteristic of some distributions derived from Debian Linux that use uninitialized memory as a source of entropy because the value stored in `junk` is indeterminate. However, because accessing an indeterminate value is undefined behavior, compilers may optimize out the uninitialized variable access completely, leaving only the time and process ID and resulting in a loss of desired entropy.

```
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>

void func(void) {
  struct timeval tv;
  unsigned long junk;

  gettimeofday(&tv, NULL);
  srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ junk);
}
```

In security protocols that rely on unpredictability, such as RSA encryption, a loss in entropy results in a less secure system.

Compliant Solution (POSIX, Entropy)

This compliant solution seeds the random number generator by using the CPU clock and the real-time clock instead of reading uninitialized memory:

```
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <sys/time.h>

void func(void) {
  double cpu_time;
  struct timeval tv;

  cpu_time = ((double) clock()) / CLOCKS_PER_SEC;
  gettimeofday(&tv, NULL);
  srandom((getpid() << 16) ^ tv.tv_sec ^ tv.tv_usec ^ cpu_time);
}
```

Noncompliant Code Example (`realloc()`)

The `realloc()` function changes the size of a dynamically allocated memory object. The initial `size` bytes of the returned memory object are unchanged, but any newly added space is uninitialized, and its value is indeterminate. As in the case of `malloc()`, accessing memory beyond the size of the original object is undefined behavior 181.

It is the programmer's responsibility to ensure that any memory allocated with `malloc()` and `realloc()` is properly initialized before it is used.

In this noncompliant code example, an array is allocated with `malloc()` and properly initialized. At a later point, the array is grown to a larger size but not initialized beyond what the original array contained. Subsequently accessing the uninitialized bytes in the new array is undefined behavior.

```
#include <stdlib.h>
#include <stdio.h>
enum { OLD_SIZE = 10, NEW_SIZE = 20 };

int *resize_array(int *array, size_t count) {
  if (0 == count) {
    return 0;
  }

  int *ret = (int *)realloc(array, count * sizeof(int));
  if (!ret) {
    free(array);
    return 0;
  }

  return ret;
}

void func(void) {

  int *array = (int *)malloc(OLD_SIZE * sizeof(int));
  if (0 == array) {
    /* Handle error */
  }

  for (size_t i = 0; i < OLD_SIZE; ++i) {
    array[i] = i;
  }

  array = resize_array(array, NEW_SIZE);
  if (0 == array) {
    /* Handle error */
  }

  for (size_t i = 0; i < NEW_SIZE; ++i) {
    printf("%d ", array[i]);
  }
}
```

Compliant Solution (`realloc()`)

In this compliant solution, the `resize_array()` helper function takes a second parameter for the old size of the array so that it can initialize any newly allocated elements:

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

enum { OLD_SIZE = 10, NEW_SIZE = 20 };

int *resize_array(int *array, size_t old_count, size_t new_count) {
  if (0 == new_count) {
    return 0;
  }

  int *ret = (int *)realloc(array, new_count * sizeof(int));
  if (!ret) {
    free(array);
    return 0;
  }

  if (new_count > old_count) {
    memset(ret + old_count, 0, (new_count - old_count) * sizeof(int));
  }

  return ret;
}

void func(void) {

  int *array = (int *)malloc(OLD_SIZE * sizeof(int));
  if (0 == array) {
    /* Handle error */
  }

  for (size_t i = 0; i < OLD_SIZE; ++i) {
    array[i] = i;
  }

  array = resize_array(array, OLD_SIZE, NEW_SIZE);
  if (0 == array) {
    /* Handle error */
  }

  for (size_t i = 0; i < NEW_SIZE; ++i) {
    printf("%d ", array[i]);
  }
}
```

Exceptions

EXP33-C-EX1: Reading uninitialized memory by an lvalue of type `unsigned char` does not trigger undefined behavior. The `unsigned char` type is defined to not have a trap representation, which allows for moving bytes without knowing if they are initialized. (See the C Standard, 6.2.6.1, paragraph 3.) However, on some architectures, such as the Intel Itanium, registers have a bit to indicate whether or not they have been initialized. The C Standard, 6.3.2.1, paragraph 2, allows such implementations to cause a trap for an object that never had its address taken and is stored in a register if such an object is referred to in any way.

Risk Assessment

Reading uninitialized variables is undefined behavior and can result in unexpected program behavior. In some cases, these security flaws may allow the execution of arbitrary code.

Reading uninitialized variables for creating entropy is problematic because these memory accesses can be removed by compiler optimization. VU#925211 is an example of a vulnerability caused by this coding error.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP33-C	High	Probable	Medium	P12	L1

Automated Detection

Tool	Version	Checker	Description
Astrée	19.04	**uninitialized-local-read** **uninitialized-variable-use**	Fully checked
Axivion Bauhaus Suite	6.9.0	**CertC-EXP33**	
CodeSonar	5.1p0	**LANG.MEM.UVAR**	Uninitialized variable
Compass/ROSE			Automatically detects simple violations of this rule, although it may return some false positives. It may not catch more complex violations, such as initialization within functions taking uninitialized variables as arguments. It does catch the second noncompliant code example, and can be extended to catch the first as well
Coverity	2017.07	**UNINIT**	Implemented
Cppcheck	1.66	**uninitvar** **uninitdata** **uninitstring** **uninitMemberVar** **uninitStructMember**	Detects uninitialized variables, uninitialized pointers, uninitialized struct members, and uninitialized array elements (However, if one element is initialized, then cppcheck assumes the array is initialized.) There are FN compared to some other tools because Cppcheck tries to avoid FP in impossible paths.
GCC	4.3.5		Can detect some violations of this rule when the `-Wuninitialized` flag is used
Klocwork	2018	**UNINIT.HEAP.MIGHT** **UNINIT.HEAP.MUST** **UNINIT.STACK.ARRAY.MIGHT** **UNINIT.STACK.ARRAY.MUST UNINIT.STACK.ARRAY.PARTIAL.MUST** **UNINIT.STACK.MIGHT** **UNINIT.STACK.MUST**	
LDRA tool suite	9.7.1	**53 D, 69 D, 631 S, 652 S**	Fully implemented
Parasoft C/C++test	10.4.2	**CERT_C-EXP33-a**	Avoid use before initialization
Parasoft Insure++	10.4.2		Runtime analysis
Polyspace Bug Finder	R2019a	CERT C: Rule EXP33-C	Checks for: • Non-initialized variable • Non-initialized pointer Rule partially covered
PRQA QA-C	9.5	**2726, 2727, 2728, 2961, 2962, 2963, 2966,** **2967, 2968, 2971, 2972, 2973, 2976, 2977,** **2978**	Fully implemented
PRQA QA-C++	4.3	**2961, 2962, 2963, 2966, 2967, 2968, 2971,** **2972, 2973, 2976, 2977, 2978**	
PVS-Studio	6.23	**V573**, **V614**, **V670**, **V679**	
RuleChecker	19.04	**uninitialized-local-read**	Partially checked
Splint	3.1.1		
TrustInSoft Analyzer	1.38	**initialisation**	Exhaustively verified (see one compliant and one non-compliant example).

Related Vulnerabilities

CVE-2009-1888 results from a violation of this rule. Some versions of SAMBA (up to 3.3.5) call a function that takes in two potentially uninitialized variables involving access rights. An attacker can exploit these coding errors to bypass the access control list and gain access to protected files [xorl 2009].

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	MSC00-C. Compile cleanly at high warning levels	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C Secure Coding Standard	MSC01-C. Strive for logical completeness	Prior to 2018-01-12: CERT: Unspecified Relationship
CERT C	EXP53-CPP. Do not read uninitialized memory	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TR 24772:2013	Initialization of Variables [LAV]	Prior to 2018-01-12: CERT: Unspecified Relationship
ISO/IEC TS 17961	Referencing uninitialized memory [uninitref]	Prior to 2018-01-12: CERT: Unspecified Relationship
CWE 2.11	CWE-456	2017-07-05: CERT: Exact
CWE 2.11	CWE-457	2017-07-05: CERT: Exact
CWE 2.11	CWE-758	2017-07-05: CERT: Rule subset of CWE
CWE 2.11	CWE-908	2017-07-05: CERT: Rule subset of CWE

CERT-CWE Mapping Notes

Key here for mapping notes

CWE-119 and EXP33-C

- Intersection(CWE-119, EXP33-C) = Ø

- EXP33-C is about reading uninitialized memory, but this memory is considered part of a valid buffer (on the stack, or returned by a heap function). No buffer overflow is involved.

CWE-676 and EXP33-C

- Intersection(CWE-676, EXP33-C) = Ø

- EXP33-C implies that memory allocation functions (e.g., malloc()) are dangerous because they do not initialize the memory they reserve. However, the danger is not in their invocation, but rather reading their returned memory without initializing it.

CWE-758 and EXP33-C

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

CWE-758 = Union(EXP33-C, list) where list =

- Undefined behavior that results from anything other than reading uninitialized memory

CWE-665 and EXP33-C

Intersection(CWE-665, EXP33-C) = Ø

CWE-665 is about correctly initializing items (usually objects), not reading them later. EXP33-C is about reading memory later (that has not been initialized).

CWE-908 and EXP33-C

CWE-908 = Union(EXP33-C, list) where list =

- Use of uninitialized items besides raw memory (objects, disk space, etc)

New CWE-CERT mappings:

CWE-123 and EXP33-C

Intersection(CWE-123, EXP33-C) = Ø

EXP33-C is only about reading uninitialized memory, not writing, whereas CWE-123 is about writing.

CWE-824 and EXP33-C

EXP33-C = Union(CWE-824, list) where list =

- Read of uninitialized memory that does not represent a pointer

Bibliography

[Flake 2006]	
[ISO/IEC 9899:2011]	Subclause 6.7.9, "Initialization" Subclause 6.2.6.1, "General" Subclause 6.3.2.1, "Lvalues, Arrays, and Function Designators"
[Mercy 2006]	
[VU#925211]	
[Wang 2012]	"More Randomness or Less"
[xorl 2009]	"CVE-2009-1888: SAMBA ACLs Uninitialized Memory Read"