EXP53-CPP. Do not read uninitialized memory

Local, automatic variables assume unexpected values if they are read before they are initialized. The C++ Standard, [dcl.init], paragraph 12 [ISO/IEC 14882-2014], states the following:

> *If no initializer is specified for an object, the object is default-initialized. When storage for an object with automatic or dynamic storage duration is obtained, the object has an* indeterminate value*, and if no initialization is performed for the object, that object retains an indeterminate value until that value is replaced. If an indeterminate value is produced by an evaluation, the behavior is undefined except in the following cases:*
>
> *— If an indeterminate value of unsigned narrow character type is produced by the evaluation of:*
> *— the second or third operand of a conditional expression,*
> *— the right operand of a comma expression,*
> *— the operand of a cast or conversion to an unsigned narrow character type, or*
> *— a discarded-value expression,*
> *then the result of the operation is an indeterminate value.*
> *— If an indeterminate value of unsigned narrow character type is produced by the evaluation of the right operand of a simple assignment operator whose first operand is an lvalue of unsigned narrow character type, an indeterminate value replaces the value of the object referred to by the left operand.*
> *— If an indeterminate value of unsigned narrow character type is produced by the evaluation of the initialization expression when initializing an object of unsigned narrow character type, that object is initialized to an indeterminate value.*

The default initialization of an object is described by paragraph 7 of the same subclause:

> *To* default-initialize *an object of type T means:*
> *— if T is a (possibly cv-qualified) class type, the default constructor for T is called (and the initialization is ill-formed if T has no default constructor or overload resolution results in an ambiguity or in a function that is deleted or inaccessible from the context of the initialization);*
> *— if T is an array type, each element is default-initialized;*
> *— otherwise, no initialization is performed.*
> *If a program calls for the default initialization of an object of a const-qualified type T, T shall be a class type with a user-provided default constructor.*

As a result, objects of type T with automatic or dynamic storage duration must be explicitly initialized before having their value read as part of an expression unless T is a class type or an array thereof or is an unsigned narrow character type. If T is an unsigned narrow character type, it may be used to initialize an object of unsigned narrow character type, which results in both objects having an indeterminate value. This technique can be used to implement copy operations such as `std::memcpy()` without triggering undefined behavior.

Additionally, memory dynamically allocated with a `new` expression is default-initialized when the *new-initialized* is omitted. Memory allocated by the standard library function `std::calloc()` is zero-initialized. Memory allocated by the standard library function `std::realloc()` assumes the values of the original pointer but may not initialize the full range of memory. Memory allocated by any other means (`std::malloc()`, allocator objects, `operator new()`, and so on) is assumed to be default-initialized.

Objects of static or thread storage duration are zero-initialized before any other initialization takes place [ISO/IEC 14882-2014] and need not be explicitly initialized before having their value read.

Reading uninitialized variables for creating entropy is problematic because these memory accesses can be removed by compiler optimization. VU925211 is an example of a vulnerability caused by this coding error [VU#925211].

Noncompliant Code Example

In this noncompliant code example, an uninitialized local variable is evaluated as part of an expression to print its value, resulting in undefined behavior.

```
#include <iostream>

void f() {
  int i;
  std::cout << i;
}
```

Compliant Solution

In this compliant solution, the object is initialized prior to printing its value.

```
#include <iostream>

void f() {
  int i = 0;
  std::cout << i;
}
```

Noncompliant Code Example

In this noncompliant code example, an `int *` object is allocated by a *new-expression*, but the memory it points to is not initialized. The object's pointer value and the value it points to are printed to the standard output stream. Printing the pointer value is well-defined, but attempting to print the value pointed to yields an indeterminate value, resulting in undefined behavior.

```
#include <iostream>

void f() {
  int *i = new int;
  std::cout << i << ", " << *i;
}
```

Compliant Solution

In this compliant solution, the memory is direct-initialized to the value `12` prior to printing its value.

```
#include <iostream>

void f() {
  int *i = new int(12);
  std::cout << i << ", " << *i;
}
```

Initialization of an object produced by a *new-expression* is performed by placing (possibly empty) parenthesis or curly braces after the type being allocated. This causes direct initialization of the pointed-to object to occur, which will zero-initialize the object if the initialization omits a value, as illustrated by the following code.

```
int *i = new int(); // zero-initializes *i
int *j = new int{}; // zero-initializes *j
int *k = new int(12); // initializes *k to 12
int *l = new int{12}; // initializes *l to 12
```

Noncompliant Code Example

In this noncompliant code example, the class member variable `c` is not explicitly initialized by a *ctor-initializer* in the default constructor. Despite the local variable `s` being default-initialized, the use of `c` within the call to `S::f()` results in the evaluation of an object with indeterminate value, resulting in undefined behavior.

```
class S {
  int c;

public:
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

Compliant Solution

In this compliant solution, S is given a default constructor that initializes the class member variable c.

```
class S {
  int c;

public:
  S() : c(0) {}
  int f(int i) const { return i + c; }
};

void f() {
  S s;
  int i = s.f(10);
}
```

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.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP53-CPP	High	Probable	Medium	**P12**	**L1**

Automated Detection

Tool	Version	Checker	Description
Clang	3.9	`-Wuninitialized` `clang-analyzer-core.UndefinedBinaryOperatorResult`	Does not catch all instances of this rule, such as uninitialized values read from heap-allocated memory.
CodeSonar	5.2p0	**LANG.STRUCT.RPL** **LANG.MEM.UVAR**	Return pointer to local Uninitialized variable
Klocwork	2018	**UNINIT.CTOR.MIGHT** **UNINIT.CTOR.MUST** **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**	Partially implemented
Parasoft C /C++test	10.4.2	**CERT_CPP-EXP53-a**	Avoid use before initialization
Parasoft Insure++			Runtime detection
Polyspace Bug Finder	R2019b	CERT C++: EXP53-CPP	Checks for: • Non-initialized variable • Non-initialized pointer Rule partially covered.
PRQA QA-C++	4.4	**2726, 2727, 2728, 2961, 2962, 2963, 2966, 2967, 2968, 2971, 2972, 2973, 2976, 2977, 2978**	
PVS-Studio	6.23	**V546**, **V573**, **V614**, **V670**, **V679**, **V730**, **V788**, **V1007**	

Related Vulnerabilities

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

Related Guidelines

SEI CERT C Coding Standard	EXP33-C. Do not read uninitialized memory

Bibliography

[ISO/IEC 14882-2014]	Clause 5, "Expressions" Subclause 5.3.4, "New" Subclause 8.5, "Initializers" Subclause 12.6.2, "Initializing Bases and Members"
[Lockheed Martin 2005]	Rule 142, All variables shall be initialized before use