EXP58-CPP. Pass an object of the correct type to va_start

While rule DCL50-CPP. Do not define a C-style variadic function forbids creation of such functions, they may still be defined when that function has external, C language linkage. Under these circumstances, care must be taken when invoking the `va_start()` macro. The C-standard library macro `va_start()` imposes several semantic restrictions on the type of the value of its second parameter. The C Standard, subclause 7.16.1.4, paragraph 4 [ISO/IEC 9899:2011], states the following:

> *The parameter* parmN *is the identifier of the rightmost parameter in the variable parameter list in the function definition (the one just before the `...`). If the parameter* parmN *is declared with the `register` storage class, with a function or array type, or with a type that is not compatible with the type that results after application of the default argument promotions, the behavior is undefined.*

These restrictions are superseded by the C++ Standard, [support.runtime], paragraph 3 [ISO/IEC 14882-2014], which states the following:

> *The restrictions that ISO C places on the second parameter to the `va_start()` macro in header `<stdarg.h>` are different in this International Standard. The parameter `parmN` is the identifier of the rightmost parameter in the variable parameter list of the function definition (the one just before the `...`). If the parameter `parmN` is of a reference type, or of a type that is not compatible with the type that results when passing an argument for which there is no parameter, the behavior is undefined.*

The primary differences between the semantic requirements are as follows:

- You must not pass a reference as the second argument to `va_start()`.
- Passing an object of a class type that has a nontrivial copy constructor, nontrivial move constructor, or nontrivial destructor as the second argument to `va_start` is conditionally supported with implementation-defined semantics ([expr.call] paragraph 7).
- You may pass a parameter declared with the `register` keyword ([dcl.stc] paragraph 3) or a parameter with a function type.

Passing an object of array type still produces undefined behavior in C++ because an array type as a function parameter requires the use of a reference, which is prohibited. Additionally, passing an object of a type that undergoes default argument promotions still produces undefined behavior in C++.

Noncompliant Code Example

In this noncompliant code example, the object passed to `va_start()` will undergo a default argument promotion, which results in undefined behavior.

```
#include <cstdarg>

extern "C" void f(float a, ...) {
  va_list list;
  va_start(list, a);
  // ...
  va_end(list);
}
```

Compliant Solution

In this compliant solution, `f()` accepts a `double` instead of a `float`.

```
#include <cstdarg>

extern "C" void f(double a, ...) {
  va_list list;
  va_start(list, a);
  // ...
  va_end(list);
}
```

Noncompliant Code Example

In this noncompliant code example, a reference type is passed as the second argument to `va_start()`.

```
#include <cstdarg>
#include <iostream>

extern "C" void f(int &a, ...) {
  va_list list;
  va_start(list, a);
  if (a) {
    std::cout << a << ", " << va_arg(list, int);
    a = 100; // Assign something to a for the caller
  }
  va_end(list);
}
```

Compliant Solution

Instead of passing a reference type to `f()`, this compliant solution passes a pointer type.

```
#include <cstdarg>
#include <iostream>

extern "C" void f(int *a, ...) {
  va_list list;
  va_start(list, a);
  if (a && *a) {
    std::cout << a << ", " << va_arg(list, int);
    *a = 100; // Assign something to *a for the caller
  }
  va_end(list);
}
```

Noncompliant Code Example

In this noncompliant code example, a class with a nontrivial copy constructor (`std::string`) is passed as the second argument to `va_start()`, which is conditionally supported depending on the implementation.

```
#include <cstdarg>
#include <iostream>
#include <string>

extern "C" void f(std::string s, ...) {
  va_list list;
  va_start(list, s);
  std::cout << s << ", " << va_arg(list, int);
  va_end(list);
}
```

Compliant Solution

This compliant solution passes a `const char *` instead of a `std::string`, which has well-defined behavior on all implementations.

```
#include <cstdarg>
#include <iostream>

extern "C" void f(const char *s, ...) {
  va_list list;
  va_start(list, s);
  std::cout << (s ? s : "") << ", " << va_arg(list, int);
  va_end(list);
}
```

Risk Assessment

Passing an object of an unsupported type as the second argument to `va_start()` can result in undefined behavior that might be exploited to cause data integrity violations.

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
EXP58-CPP	Medium	Unlikely	Medium	**P4**	**L3**

Automated Detection

Tool	Version	Checker	Description
Clang	3.9	`-Wvarargs`	Does not catch the violation in the third noncompliant code example (it is conditionally supported by Clang)
Parasoft C/C++test	10.4.2	**CERT_CPP-EXP58-a**	Use macros for variable arguments correctly
Polyspace Bug Finder	R2019b	CERT C++: EXP58-CPP	Checks for incorrect data types for second argument of va_start (rule fully covered)

Related Vulnerabilities

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

Related Guidelines

SEI CERT C++ Coding Standard	DCL50-CPP. Do not define a C-style variadic function

Bibliography

[ISO/IEC 9899:2011]	Subclause 7.16.1.4, "The `va_start` Macro"
[ISO/IEC 14882-2014]	Subclause 18.10, "Other Runtime Support"