Document number:P0799R0

Date: 20171016 (pre-Albuquerque)
Project: Programming Language C++, WG21, SG12
Authors: Stephen Michel, Chris Szalwinski, Michael Wong, Hubert Tong,
Email: stephen.michell@maurya.on.ca, chris.szalwinski@gmail.com, michael@codeplay.com, hubert.reinterpretcast@gmail.com
Reply to: stephen.michell@maurya.on.ca

Programming vulnerabilities for C++ (part of WG23 N0746)

Introduction

After discussions with WG 21/SG 12, WG 23 experts augmented by Canadian C++ experts have commenced a process to prepare a vulnerability document for review by WG 21.

At its meeting in July, WG 23, and at its quarterly meeting in September, the Canadian mirror committee WG 21 and WG 23 worked on individual vulnerability descriptions for this document. The relevant sub-clauses worked on by this group are included below, specifically

- 6.5 Enumerator issues [CCB],
- 6.13 Null pointer dereference [XYH],
- 6.22 Initialization of variables [LAV], and
- 6.39 Deep vs shallow copying [YAN]

These clauses are not complete, but contain the thinking of the participants. It tis the wish of the group to seek feedback from WG21 SG12 on these submitted vulnerabilities for possible inclusion, or modification in the C++ Core Guidelines

WG 23 will also meet in Albuquerque to discuss and further develop this document alongside WG 21, as well as on Webex. Some of the time will be joint with WG 21/SG 12.

The full document can be accessed at www.open-std.org/jtc1/sc22/wg23/ and is document N0744.

Target audience of this document

From clause 4.2 of TR 24772-1

> The intended audience for this document are those who are concerned with assuring the predictable execution of the software of their system; that is, those who are developing, qualifying, or maintaining a software system and need to avoid language constructs that could cause the software to execute in a manner other than intended.

Developers of applications that have clear safety, security or mission-criticality are expected to be aware of the risks associated with their code and could use this document to ensure that their development practices address the issues presented by the chosen programming languages, for example by subsetting or providing coding guidelines.

It should not be assumed, however, that other developers can ignore this Technical Report. A weakness in a non-critical application may provide the route by which an attacker gains control of a system or otherwise disrupts co-hosted applications that are critical. It is hoped that all developers would use this Technical Report to ensure that common vulnerabilities are removed or at least minimized from all applications.

Specific audiences for this document include developers, maintainers and regulators of:

- Safety-critical applications that might cause loss of life, human injury, or damage to the environment.
- Security-critical applications that must ensure properties of confidentiality, integrity, and availability.
- Mission-critical applications that must avoid loss or damage to property or finance.
- Business-critical applications where correct operation is essential to the successful operation of the business.
- Scientific, modeling and simulation applications that require high confidence in the results of possibly complex, expensive and extended calculation.

Expressed another way, this document does not intend to create coding guidelines for organizations. The expectation is that the senior technical people, such as those that analyse the safety and security risks and create coding standards for the organization will use the guidance of these documents to create the appropriate coding standards for the sector, organization and the product.

Relationship with C++ Core Guidelines

TR 24772-9 C++ Programming Language Vulnerabilities is intended to complement and support the work being done in the C++ Core Guidelines. Where vulnerabilities have been identified that overlap guidance from the Core guidelines, these will be referenced. We trust that the core guidelines will similarly leverage the guidance of this document in providing guidance.

Work in Progress

The format of the remainder of document matches that of N0744, where there is a language independent section, and language-specific versions. We skip Section 1-5 as well as other sections at this time where we say it is not relevant to the discussion yet.

1. Scope

Clauses 1-5.

Not relevant at this time**, except**
Explaining the differences between C and C++ in the existence of vulnerabilities. Either clause 5 or clause 6.1 needs state that C++ contains C as a subset, and hence if you use the C subset, you are vulnerable to the C vulnerabilities as documented in TR 24772-3.

6. Specific Guidance for C++ Vulnerabilities

6.1 General

Not relevant at this time

6.2 Type System [IHN]

Not relevant at this time

6.3 Bit Representations [STR]

Not relevant at this time

6.4 Floating-point Arithmetic [PLF]

Not relevant at this time

6.5 Enumerator Issues [CCB]

6.5.1 Applicability to language

C++ offers *scoped enums* to replace the *enum* capability of C. Even if one uses C-stlye *enums*, C++ forbids implicit casts from integer to an enum, therefore preventing A = B+C, where A, B and C are variables of the same enum.

In C++, there is not a bidirectional cast between objects of an **enum class** and **int**, i.e. there is no implicit conversion from an integer type back to the enum type, hence operations such as "++", "+", "<" and enumerations used as array indexes are unavailable unless explicitly declared in the program. Hence, the general vulnerability of operations on enumerations, such as "+" and "*" is avoided. Note, however, X+Y could become an integer that is the controlling operand to an overloaded function.

Idea that the enumerated type can have a user-specified underlying type for enumerated constants

Discussion - Chris's approach:
C++ offers enums for defining distinct types composed of sets of related named constants. The type of each enum is different from all other types. Each enum has an underlying type, which the user can specify, and is an int by default.

Since enums are distinct types, the user can only assign values to an object of enumerated type that are values of that enumerated type. C++ does not support implicit conversion of an int to an enum, therefore preventing A = B + C where A, B and C are variables of the same enum.

C++ enums can be scoped (enum class) or unscoped (enum). C++ supports implicit conversion of an unscoped enum to an integer by integral promotion

```
enum Color {red, green, blue};

int i = red; // implicit conversion
```

C++ does not support implicit conversion of a scoped enum to an int. That is, there is no bidirectional cast between objects of a scoped enum and int. Hence, operations such as ++, +, < and enums used as array indices require explicit definitions.

```
enum class Color {red, green, blue};

int i = red; // error – no implicit conversion
```

Additionally, C++ enum classes may overload enumeration literals from different classes, hence

enum class Color {reg, green, blue};

and

enum class Light_Color {red, yellow, green)

may exist in the same program with no ambiguity.

// Is there anything to say about data objects indexed by enums?

6.5.2 Guidance to language users

- Use *scoped enumerations* in preference to the C-style *unscoped enumerations* for related values.
  - See the guidance of <CPP Core Guidelines Enum.4 and Enum.6 …>
- Use constexpr to declare a set of unrelated values, such as

```
constexpr size_t bufferLen  = 128;
constexpr char   special_char = 'a';
```

- If *unscoped enumerations* are used, follow the general advice of TR 24772-3 clause 6.5.2 as well as the following:
  - Avoid casting arbitrary integer values to enumeration type. If it is unavoidable, use a function-style cast with braces instead of C-style or static casts

```
e_type{7};
```

  - Obtain the underlying enumeration value, by casting the enumeration to its underlying type, e.g.,

```
enum e_type{A, B, C};
auto value = static_cast<typename std::underlying_type<e_type>::type>(B);
```

6.6 Conversion Errors [FLC] - 6.12 Pointer Arithmetic [RVG]

Not relevant at this time

6.13 NULL Pointer Dereference [XYH]

This subclause was reworked 5 Sep 2017. A thorough review is required.

6.13.1 Applicability to language

The vulnerability as described in TR 24772-1 clause 6.13 exists in C++,...

C++ provides a number of mechanisms that allow the programmer to create, manipulate and destroy objects without the explicit use of raw pointers.

a) Containers manage memory and separate memory management from the use of objects.
b) The container interface throws an exception if any container cannot be allocated.
c) Smart pointer creation functions allocate heap memory and handle memory management.

C++ mechanisms "new" throws an exception if the allocated object cannot be created (i.e. if a null ptr would be returned).

6.13.2 Guidance to language users

- Avoid the use of direct memory allocation as ...
- Create a specific check that a pointer is not null before dereferencing it.
- Do not suppress exceptions on memory allocation. If exceptions are suppressed, follow the guidance of TR 24772-3 clause 6.14.2.

6.14 Dangling Reference to Heap [XYK] through 6.21 Namespace Issues [BJL]

Not relevant at this time

6.22 Initialization of Variables [LAV]

6.22.1 Applicability to language

The vulnerability as described in TR 24772-1 exists in C++.

C++ provides language capabilities to mitigate the effects of uninitialized variables as follows:

a. Provides default member initialization of the form

```
class C {

int a {6} ;    // explain
```

```
}
```

b. Constructor initialization permits the initialization of all members of a class, however, there is no coverage check

c.

For Discussion - Chris' suggestion:

The vulnerability as described in TR 24772-1 exists in C++.

C++ provides multiple language capabilities to mitigate the effects of uninitialized variables as follows:

a. Default member initialization through in-class declaration

```
class C {
    int a {6};         // default
    int* p {nullptr}; // default
public:
     C() = default;     // all variables default initialized
     C(int a_):a{a_}(); // p - default
     C(const C& src) { *this = src; }
     // ...
};
```

b. Member initialization initializes individual instance variables, however, there is no coverage check

```
class C {
    int* p;
    int a;
    int b; // not initialized
  public:
    C(): p{nullptr},a{6} {}
    C(const C& src): p{nullptr} { *this = src; }
    // ...
};
```

c. Constructor initialization initializes an instance variable before use; however, there is no coverage check and no assurance that immediately upon construction all variables have been initialized

```
class C {
    int* p;
    int a;
    int b; // not used in ctor, hence not initialized
  public:
    C(){ a = 6; p = nullptr; }
    C(const C& src): { p = nullptr; *this = src; }
         // ...
};
```

6.22.2 Guidance to language users

- Follow the guidance of TR 24772-1 clause 6.22.5.
- Others?

For Discussion - Chris' suggestion:

- Always initialize instance variables to default values at declaration
    - See the guidance of <CPP Core Guidelines ES.20>
- If any instance variable is not initialized at declaration, member initialize that instance variable.
- If any instance variable is not member initialized or initialized to a default value at declaration, initialize that instance variable in each constructor's logic.
- If any instance variable has not been initialized once the constructor has completed execution, follow the guidance of TR 24772-1 clause 6.22.5.

6.23 Operator Precedence and Associativity [JCW] through 6.37 Type-breaking Reinterpretation of Data [AMV]

Not relevant at this time

6.38 Deep vs. Shallow Copying [YAN]

6.38.1 Applicability to Language

This vulnerability exists in C++ as described in TR 24772-1 clause 6.39.

Since C++ is often used by new programmers coming from C, they may look to the calling code of a function to determine if parameters are passed by value or by reference. Since C++ has reference parameters, the address of the argument may escape without the explicit use of the address operator.

C++ provides the "string view" mechanism as safer pointers to strings. Updates through view are prohibited, but the initial non "view" value can be updated and this change will be seen by all viewers, even if they are dependent on fixed value.

6.38.2 Guidance to language users

- Follow the mitigations of TR 24772-1 clause 6.9.5.
- Avoid updating the value of a string while there are valid string views in existence.

6.39 Memory Leak [XYL] through 6.64 Uncontrolled Format String [SHL]

Not relevant at this time

7. Language specific vulnerabilities for C

[TBD]

8. Implications for standardization

TBD