Scientific and Practical Cyber Security Journal (SPCSJ) 8(4): 106 – 109 ISSN 2587-4667
Scientific Cyber Security Association (SCSA)

STATE OF MEMORY SAFETY IN C++

René Pfeiffer
DeepSec GmbH / University of Applied Sciences Technikum Wien

ABSTRACT: The C++ programming language shares its ancestry with C, but it is a language on its own. Memory safety has always been a challenge, but recently government bodies addressed defects in software applications and recommended a strategy for securing memory access. The C++ standard started to adopt a security stance beginning with C++11. Later C++ specifications improved the language further. Modern C++ includes all features to develop memory-safe software applications provided the language specification C++11 and later is used.

KEYWORDS*: C++, memory safety, secure coding, C++ standards, information security, software development, data ownership, secure design*

OVERCIEW OF MODERN C++

Modern C++ consists of the C++11 and all later language specifications (C++ standards). (Stroustrup et. al. 2015) The specifications eliminate ambiguities and clearly define how the language handles undefined behavior. Furthermore, the language features allow to track dynamic memory allocations by use of smart pointers and automatically deallocate memory blocks that go out of scope. These features require the cooperation of the developers, because the C++ constructs must be actively used. Smart pointers change the approach on how to implement memory operations. The new approach does not require the use of the operators *new[]* and *delete[]*. In fact, Modern C++ should completely avoid using these operators, because they are implicitly used by the smart pointers. The language features of Modern C++ for implementing secure code consist of the Resource Acquisition Is Initialization (RAII), object lifetime, scope exit, stack unwinding, constant declarations/expressions, and smart pointers. (Gregoire 2021)

- RAII ensures that objects stay in scope and that constructors/destructors are always called.
- Scopes consist of functions, member functions, and localized blocks of code.
- Function parameters and variables declared as *const* introduce immutable data structures.
- No use of raw pointers, *new[]* or *delete[]*.
- No use of references in functions.
- Use of unique and shared pointers with reference counting; creation must be implemented by calling *make_unique<>()* and *make_shared<>()*.
- Operations on unique pointers require move semantics to transfer ownership to code operating on data structures.

More specific advice for safety-critical code adds ten more rules for specific use cases. Among them is to avoid heap memory allocations, not using the preprocessor, and limit the use of references. These rules are known as The Power of 10. (Holzmann 2006) They are older than Modern C++ and can be applied to any programming language.

STATIC AND DYNAMIC ANALYSIS OF CODE

Bjarne Stroustrup, the creator of C++, recommends using static analysis for all code. (Stroustrup 2022). The purpose is to periodically check for errors and code violating core guidelines and best practices. Static and dynamic analysis are a standard tool in software development in order to maintain and test the quality of the code. This analysis stage can catch bugs and undesired behavior of applications, but it is limited by Rice's Theorem. (Rice 1953) The theorem states that non-trivial and extensional code (expressed by functions) processing input data is undecidable. This means that there can be no complete formal test to check if the code behaves in a well-defined manner and as expected for all variations of

input data. The theorem limits the claims that any analysis tool can make about any application code. Therefore the selection and design of the unit and security tests for the application must be carefully created. Trivial tests should be avoided. Best practice is to test for the normal and exceptional operational situations. The latter is the primary task of security tests. Software quality and security is therefore mainly defined by the set of tests performed.

FURTHER IMPROVING MEMORY SAFETY OF C++

The set of rules outlined by the Modern C++ standards serves as a baseline for all new code and refactoring. Memory safety can be improved by incorporating techniques from the programming languages Austral, Gel, Inko, Hylo/Val, and Vale. (Borretti 2021; Falcon and Cook 2009; Inko 2018; Racordon 2023; Ovadia 2022) The approach of the different proposals is similar, but the lead developer of Vale has compiled the most effective coding styles for protecting data structures in applications. (Ovadia 2023) The strategies are:

- Borrowless affine style for data structure
- Constraint references
- Generational references (for cases with limited memory use)
- Random generational reference
- Simplified unique borrowing

The borrowless affine style mandates to exclusively use stack objects and data referenced by unique pointers. No raw pointer and no C++ references are allowed. Raw arrays must be replaced by the *std::array* C++ STL container. When reading data it is always moved to the code that uses it. This ensures that the code acting on the data always has the full ownership and that all other code has no read and write access to the data. The C++ move operations utilize unique pointers in order to save expensive memory copy operations. The Rust programming language uses the same concepts.

Constraint references are implemented by explicitly adding the reference count into an object. The reference itself work similar to a shared pointer. The object has to check its reference counter and asserts a zero value once it gets out of scope. Any deconstructor can only deallocate the instance if there is no other reference to it. This protection mechanism is modeled after the foreign constraints in the Structured Query Language (SQL). The constraint references enables instances of objects to monitor its use by other parts of the code.

Generational references are an implementation of heap memory management. The design is a memory arena where created objects are referenced by their memory address and a generation number. All deallocation operations increment the generation number. All access verifies that the generation number matches the expected generation number when addressing an object instance. Generation numbers must not be reused, and the allocation process keeps the memory in order to protect it. This approach is only possible if the heap memory accommodates all storage requirements of the application at run-time, because all unreferenced memory regions are kept for protection.

Random generational reference use a pseudo-random generation number as object identifier instead of an increasing number. The purpose is to reduce the probability of generation number collisions. The random numbers act as a stochastic defense. These numbers are categorized as metadata and can be linked to the memory address of data structures. There are hardware features to support this, for example Arm v9 processors implement Memory Tagging Extensions (MTEs) for tracing memory allocations/deallocations. (Oracle® 2016, Arm Holding 2024, Serebryany 2019) Compilers developers implement memory tagging to assist with debugging. It is important to note that this technique is only useful for finding bugs. It is not recommended to build production code with it. (Clang Team 2007)

Simplified unique borrowing re-introduces mutable non-owning pointers. These pointers must never access the original object while it exists. They must never be returned by functions. They must not be stored in data structures or arrays. Also there must be no aliasing when using the pointer. A sample implementation in C++ shows how to integrate this method into existing code. (Ovadia 2023) The main reason to use simplified unique borrowing is to avoid using a full borrow checker or memory annotations such as references.

IMPLEMENTING NEW CODE

The rules can be integrated into secure coding guidelines to help with the implementation. Using the rules requires discipline, because the memory protection techniques add extra code constructs and change the way traditional C++ code was designed. This is especially important for the C++ move semantics. Testing also requires specific code to test for memory safety violations. The effort is less than a complete reimplementation in a new programming language.
The compiler tool chains can support the development process. Compilers have added check features and assistance by taking advantage of processor features. (Wei 2021) This is only a measure to support the implementation. The code must be written in a manner to enable the compilers to use the protection and debugging features.

CONVERTING EXISTING CODE

The techniques can be gradually integrated into existing code. The C++ code needs to be reduced or adapted to the language features described in this document. The most important step is to introduce unique pointers and move semantics. References may be added if the processor has hardware support and the compiler toolchain can use these features. As a bonus, all techniques can be implement in a concurrency-safe way. The borrowless affine style is already concurrency-safe. The necessary C++ standards are available in all compilers for standards platforms. The first step should always be to port the code to Modern C++. The use of generational references should be checked with the availability of the heap memory in the target system.
The testing phase can take advantage of all standard test methods for memory safety. This is an advantage, because no adaptions to the toolchain are necessary.

SUMMARY

The described memory safety techniques are available with modern C++ compilers. The methods are implemented in other programming languages. The abstraction layers and the hardware running the compiled code may introduce subtle changes in storage or execution order. All concurrent code is affected by this behavior due to the optimizations present in the run-time environment. Applications with a parallel execution design have to add synchronization techniques.
The proposed memory safety need to be evaluated for their impact. The research of the sources and related publications yielded no quantitative studies of code with a statistically significant sample size of tests. Further work is needed to test how well the proposed programming approach can counter memory safety problems with test data.

ACKNOWLEDGMENT

This work was supported by Shota Rustaveli National Science Foundation of Georgia (SRNSFG) - CG-24-220

REFERENCES

Stroustrup, Bjarne et. al. 13 September 2015. “C++ Core Guidelines”. Updated 3 October 2024. https://github.com/isocpp/CppCoreGuidelines

Gregoire, Marc. 13 February 2021. “Professional C++”. John Wiley & Sons, Inc.

Holzmann, Gerard J. June 2006. “The Power of 10: Rules for Developing Safety-Critical Code”. NASA/JPL Laboratory for Reliable Software. IEEE Computer. **39** (6): 95–99. doi:10.1109/MC.2006.212.

Stroustrup, Bjarne. December 2022. “A call to action: Think seriously about “safety”; then do something sensible about it”. Doc. no. P2739R0. Columbia University.

Rice, H. G. (1953). "Classes of recursively enumerable sets and their decision problems", Transactions of the American Mathematical Society, **74** (2): 358–366, doi:10.1090/s0002-9947-1953-0053041-6, JSTOR 1990888

Borretti, Fernando. 2021. The Austral Programming Language. Updated 2023. https://austral-lang.org/.

Falcon, J., Cook, W.R. (2009). Gel: A Generic Extensible Language . In: Taha, W.M. (eds) Domain-Specific Languages. DSL 2009. Lecture Notes in Computer Science, vol 5658. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03034-5_4

Inko development team. 2018. “The Inko programming language”. Updated October 2024. https://inko-lang.org/

Racordon, Dimitri. 2023. “The Hylo Programming Language”. Updated October 2024. https://www.hylo-lang.org/.

Ovadia, Evan. 2022. “The Vale Programming Language”. Updated October 2024. https://vale.dev/.

Ovadia, Evan. 22 June 2023. “Making C++ Memory-Safe Without Borrow Checking, Reference Counting, or Tracing Garbage Collection”. https://verdagon.dev/blog/vale-memory-safe-cpp

Oracle® Corporation. 2016. “Application Data Integrity (ADI) operations”. Updated July 2017. https://docs.oracle.com/cd/E86824_01/html/E54765/adi-2.html

Arm Holding plc. “Arm memory tagging extension”. Updated 1 November 2024. https://source.android.com/docs/security/test/memory-safety/arm-mte

Serebryany, Konstantin. 2019. “ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety”. ;login: Magazine Vol. 44 ,No. 2.

The Clang Team. 2007. “Hardware-assisted AddressSanitizer Design Documentation”. Updated 2024. https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html

Wei, Song, et. at. 2021. “A Comprehensive and Cross-Platform Test Suite for Memory Safety - Towards an Open Framework for Testing Processor Hardware Supported Security Extensions”. Computing Research Repository (CoRR). November 2021.