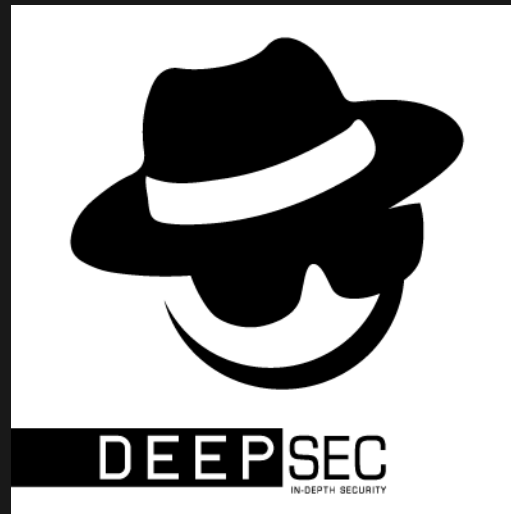












STATE OF MEMORY SAFETY IN C++

RENÉ PFEIFFER

DEEPSEC



WHOAMI

-  René „Lynx“ Pfeiffer
-  working with/for Free Software since 1988, senior systems administrator
-  [DeepSec In-Depth Security Conference](#) organisation team
-  Study of [theoretical physics](#)
-  „on the Internet“ since 1992 (14.400 baud & faster)
-  30+ years experience with software development, computing, and systems administration
-    

WHY?

Blame the NSA!



C++ IS ALMOST YOUNGER THAN RUST



- 1979 - C with Classes
- 1982 - C++, Cfront compiler
- 1984 - C++ libraries
- 1989 - C++ v2.0
- 1998 - C++98
- 2011 - C++11 (Modern C++ with bugs)
- 2014 - C++14 (Modern C++)
- 2015 - C++ Core Guidelines
- 2015 - Rust v1.0 (work started 2006)

BUFFER OVERFLOWS IN RUST

- [CVE-2021-28879](#) - Rust <1.52 standard library Zip integer/buffer overflow
- [CVE-2021-28879](#) - Rust <1.50 read_to_end() buffer overflow
- [CVE-2021-25900](#) - Rust smallvec crate SmallVec::insert_many buffer overflow
- [CVE-2020-35887](#) - Rust arr crate buffer overflow in Index and IndexMut
- [CVE-2019-15548](#) - Rust ncurses crate buffer overflow because interaction with C functions is mishandled
- [CVE-2018-1000810](#) - Integer/buffer overflow in Rust Programming Language Standard Library via str::repeat
- [CVE-2018-1000657](#) - Buffer overflow & arbitrary code execution in std::collections::vec_deque::VecDeque::reserve()
- [CVE-2017-1000430](#) - rust-base64 buffer overflow

This is **not** Rust bashing. All code can contain bugs.

RUST UNSAFE MODE

- Dereference a raw pointer
- Call an unsafe function or method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access fields of *unions*

Useful in system programming or code analysing data.

Source: [Unsafe Rust \(Unsafe Superpowers\)](#)

RUST UNSAFE MODE (PUBLICATION)

2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)

Is Rust Used Safely by Software Developers?

Ana Nora Evans
AnaNEvans@virginia.edu
University of Virginia

Bradford Campbell
bradjc@virginia.edu
University of Virginia

Mary Lou Soffa
soffa@virginia.edu
University of Virginia

Source: [Studie University of Virginia, 2020](#)

RUST UNSAFE MODE (PUBLICATION)

Rust, an emerging programming language with explosive growth, provides a robust type system that enables programmers to write memory-safe and data-race free code. To allow access to a machine's hardware and to support low-level performance optimizations, **a second language, Unsafe Rust, is embedded in Rust**. It

contains support for operations that are difficult to statically check, such as C-style pointers for access to arbitrary memory locations and mutable global variables. **When a program uses these features, the compiler is unable to statically guarantee the safety properties Rust promotes**. In this work, we perform a large-scale empirical study to explore how software developers are using Unsafe Rust in real-world Rust libraries and applications. Our results indicate that **software engineers use the keyword unsafe in less than 30% of Rust libraries, but more than half cannot be entirely statically checked by the Rust compiler because of Unsafe Rust hidden somewhere in a library's call chain**. We conclude that although the use of the keyword unsafe is limited, the propagation of unsafeness offers a challenge to the claim of Rust as a memory-safe language.

C++ PHILOSOPHY

- Light-weight abstraction
- Simple and direct mapping to hardware
- Zero-overhead abstraction mechanisms
 - You don't pay for what you don't use.
 - What you do use is just as efficient as what you could reasonably write by hand.
 - Exceptions (🙄):
 - Runtime type identification (RTTI, #include <typeinfo>)
 - Exceptions

Quelle: [Foundations of C++](#)

C++ MEMORY MANAGEMENT

- Resource Acquisition Is Initialization (RAII) Konzept
 - Scope-Bound Resource Management (SBRM)
 - Constructor ensures reservation && initialisation
 - Resource are freed by destructors
 - Use of C++ features (*object lifetime, scope exit, order of initialization, stack unwinding*)
- Garbage Collectors
 - C++ support until C++23 (removed afterwards, see [P2186R2](#))
 - Third party code - replacement of *new / delete / std::allocator* bzw. *malloc() / free()*

MEMORY SAFETY

- [CWE Kategorien](#) for Memory Safety

(Not included in slides, CWE is better maintained. )

MEMORY SAFETY & UNDEFINED BEHAVIOUR

- Memory safety is part of *undefined behaviour (UB)*
- C++11 eliminated many cases
- Post-C++11 code needs to observe [Order of Evaluation](#)
- Correct C++ code has no undefined behaviour

[C++ Reference Undefined Behaviour](#)

RAII CLASSES

- RAII-observant code - doesn't create itself!
- Function call in non-RAII classes
 - `open() / close()`
 - `lock() / unlock()`
 - `init() / copyFrom() / destroy()`

(NON-)RAII CODE

```
std::mutex m;

void non_raii()
{
    m.lock();           // Mutex anlegen
    f();                // Wenn f() eine Exception wirft, wird m nie freigegeben
    if ( not all_ok() )

        return;       // Frühzeitiges return(), Mutex bleibt
    m.unlock();        // Hier wird der Mutex freigegeben
}

// --- RAII code -----

void raii_code()
{
    std::lock_guard lk(m); // RAII Klasse: "Mutex acquisition is initialization"
    f();                  // Wenn f() eine Exception wirft, wird m freigegeben
    if ( not all_ok() )
        return();        // Frühzeitiges return(), Mutex wird freigegeben
}
```

THE POWER OF 10

1. Avoid complex flow constructs, such as *goto* and recursion.
2. All loops must have fixed bounds. This prevents runaway code.
3. Avoid heap memory allocation.
4. Restrict functions to a single printed page.
5. Use a minimum of two runtime assertions per function.
6. Restrict the scope of data to the smallest possible.
7. Check the return value of all non-void functions, or cast to void to indicate the return value is useless.
8. Use the preprocessor sparingly.
9. Limit pointer use to a single dereference, and do not use function pointers.
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

Source: [The Power of 10: Rules for Developing Safety-Critical Code](#)

THE POWER OF 10

1. Avoid complex flow constructs, such as *goto* and recursion.
2. All loops must have fixed bounds. This prevents runaway code.
3. **Avoid heap memory allocation.**
4. Restrict functions to a single printed page.
5. Use a minimum of two runtime assertions per function.
6. Restrict the scope of data to the smallest possible.
7. Check the return value of all non-void functions, or cast to void to indicate the return value is useless.
8. **Use the preprocessor sparingly (i.e. not at all!).**
9. **Limit pointer use to a single dereference, and do not use function pointers.**
10. Compile with all possible warnings active; all warnings should then be addressed before release of the software.

Source: [The Power of 10: Rules for Developing Safety-Critical Code](#)

MEMORY SAFETY GUIDELINES

- Use smart pointers (`unique_ptr`, `shared_ptr`, `weak_ptr`)
- Avoid using heap and dynamic memory
- Use RAII
- Static analysis (recommendation of Bjarne Stroustrup)
- Use Modern C++
 - Based on C++11, C++14, C++17, C++20, ...
 - Replace `new` with `std::make_unique<>()` and `std::make_shared<>()`
 - Use `const`, convert `const` to `constexpr` where possible

STATIC ANALYSIS (1)

```
#include <iostream>

using namespace std;

auto main() -> int {
    string s1 = "abcdefg";
    string_view s2 = s1;    // std::string_view introduced in C++17

    cout << s2.length() << " " << s2 << endl;
    s1.erase(2,2);
    cout << s2.length() << " " << s2 << endl;
    return(0);
}
```

Output:

```
7 abcdefg
7 abefgg
```

STATIC ANALYSIS (2)

```
clang++-17 -Wall -Wpedantic -std=c++20 -o memory_string memory_string.cc
```

```
clang++-17 -Wall -Wpedantic -std=c++20 -fsanitize=address -lasan -fuse-ld=lld-17 -o memory_string memory_string.cc
```

```
scan-build-17 clang++-17 -Wall -Wpedantic -std=c++20 -o memory_string memory_string.cc
```

```
clang-tidy-17 -p . -extra-arg=-std=c++20 -checks=boost-*,bugprone-*,clang-analyzer-*,cppcoreguidelines-*,modernize-*,cert-* \\  
memory_string.cc
```

```
clang-tidy-17 -p . -extra-arg=-std=c++20 -checks=* memory_string.cc
```

```
g++ -fanalyzer -std=c++20 -Wall -Wpedantic -o memory_string memory_string.cc
```

No warnings, no errors, no hints.

WHAT ELSE IS THERE?

- Tracing Garbage Collectors (GCs)
 - **slow**
 - „stop_world()“ issues, additional threads
- Reference counting (RC)
 - **slow (overhead)**
 - depends on memory management
- Borrow checking
 - **adds complexity**
 - C++ is sufficiently complex already

STRATEGY FOR C++

- „Borrowless affine style“, via `unique_ptr` & ownerships (from [Vale](#), [Val](#), and [Austral](#))
- Constraint references - as in SQL (from [Gel](#), [Inko](#), and [Vale](#))
- Generational references & random generational references (from [Vale](#))
- Simplified borrowing (from [Val](#))

Source: [Making C++ Memory-Safe Without Borrow Checking, Reference Counting, or Tracing Garbage Collection](#)

BORROWLESS AFFINE STYLE

- Dereferencing `unique_ptr` is safe if
 - `unique_ptr` was initialised / contains sensible data
 - `unique_ptr` has not been `std::moved`
- Stack objects are safe
 - Direct access („*whole*“ *objects*)
 - No pointer operations on stack
- Avoid raw pointers **and** references (!)
 - Use `unique_ptr` or move semantics
 - Use an index / ID for fields
 - Read fields - take ownership first, swap fields out
- Don't use raw arrays - use `std::array` (or similar collections)
- Read element by taking ownership and removing it (avoids race conditions)

GENERATIONAL REFERENCES

- All objects have a generation number (*g_ref*)
- All memory access requires object address and valid *g_ref*
- Free operations increase *g_ref*

```
// Makes an object
gowned(Ship) ship = make_gowned(42);

// Makes a generational reference to it
gref(Ship) shipRef = ship.ref();

// Does a generation check
auto shipHandle = shipRef.open();

// Prints 42
cout << shipHandle->fuel << endl;
```

Objects use heap, no real free operations (generation number stays).

RANDOM GENERATIONAL REFERENCES

- Generation number g_ref is replaced by PRNG number
- Stochastic protection against collisions
- ARM CPU Memory Tagging Extension (MTE) is similar
 - adds metadata to allocation/deallocation
 - tags memory regions
- [Sample implementation in C++](#) available from Vale developers
- **Important:** References are not part of the objects!

CONSTRAINT REFERENCES

- Put references directly into object
- Introduce `constraint_ref` (shared_ref with references)
- Objects out of scope
 - assert that reference (count) is zero
 - ensure no one is pointing at them
- Similar to foreign key constraint in databases
- *Any* constraint violations halts code
- [Sample implementation in C++](#) available from Vale developers

SIMPLIFIED UNIQUE BORROWING

- Fix swap-out access by re-adding mutable non-owning pointers with rules:
 - Never access original objects while they exist
 - Never return these pointers
 - Never store these pointers in structs/arrays
 - Never alias them
- Extending rules can re-allow `shared_ptr`
 - Swap out / protect accessed elements is required
 - Introduce const semantics for protection

Read [Making C++ Memory-Safe Without Borrow Checking, Reference Counting, or Tracing Garbage Collection](#) often! 🙄

CONCLUSION

- There is no zero-overhead memory safety for any language.
- C++ never adds implicit overhead (design philosophy).
- Modern C++ provides a complete toolbox for memory safety.
- Modern C++ and memory safe techniques can be added *gradually*.
- Programming languages need at least 10-15 years to mature.
- Know your use cases!
 - Identify and protect critical sections.
 - Never deallocating memory is memory safe, too. 😊
- Refactoring is cheaper than switching programming languages.

QUESTIONS?



Source: [Mesa Gets Patch For Official Intel Whiskey Lake Support](#)

SOURCES

- [Austral programming language](#)
- [Boehm-Demers-Weiser Garbage Collector](#)
- [C++ Best Practices](#)
- [C++ Core Guidelines](#)
- [C++ Exceptions](#)
- [Can C++ Be Saved? Bjarne Stroustrup on Ensuring Memory Safety](#)
- [CWE CATEGORY: Comprehensive Categorization: Memory Safety](#)
- [Inko programming language](#)
- [Oilpan: C++ Garbage Collection \(Chrome Project\)](#)
- [The Power of 10: Rules for Developing Safety-Critical Code](#)
- [Towards memory safety in C++ \(P2771R0\)](#)
- [Type-and-resource safety in modern C++](#)
- [What is Modern C++?](#)
- [Val programming language](#)
- [Vale programming language](#)

DOOM 3 SOURCE CODE HINTS

- *Code should be locally coherent and single-functioned: One function should do exactly one thing. It should be clear about what it's doing.*
- *Local code should explain, or at least hint at the overall system design.*
- *Code should be self-documenting. Comments should be avoided whenever possible. Comments duplicate work when both writing and reading code. If you need to comment something to make it understandable it should probably be rewritten.*

Source: [The Exceptional Beauty of Doom 3's Source Code](#)

ABOUT THE AUTHOR

René „Lynx“ Pfeiffer was born in the year of Atari's founding and the release of the game Pong. Since his early youth he started taking things apart to see how they work. He couldn't even pass construction sites without looking for electrical wires that might seem interesting. The interest in computing began when his grandfather bought him a 4-bit microcontroller with 256 byte RAM and a 4096 byte operating system, forcing him to learn Texas Instruments TMS 1600 assembler before any other programming language.

After finishing school he went to university in order to study physics. He then collected experiences with a C64, a C128, two Commodore Amigas, DEC's Ultrix, OpenVMS and finally GNU/Linux on a PC in 1997. He is using Linux since this day and still likes to take things apart und put them together again. Freedom of tinkering brought him close to the Free Software movement, where he puts some effort into the right to understand how things work – which he still does.

René is a senior systems administrator, a lecturer at the University of Applied Sciences Technikum Wien and FH Burgenland, and a senior security consultant. He uses all the skills in order to develop security architectures, maintain/improve IT infrastructure, test applications, and to analyse security-related attributes of applications, networks (wired/wireless, components), (cryptographic algorithms), protocols, servers, cloud platforms, and more indicators of modern life.