Safer C++

Safety Profiles And Their Adoption

Matthew Wadsworth

ISO/IEC 23643:2020 Defintions

- Software **security** (or “cybersecurity” or similar)
  - Making software able to **protect its assets** from a malicious attacker
  - Examples: securing power grids, hospitals, banks, personal data, secrets, …
- Software **safety** (or “life safety” or similar)
  - Making software free from unacceptable risk of **causing unintended harm** to humans, property, or the environment
  - Examples: hospital equipment, autonomous vehicles/weapons
- Programming language safety
  - A language’s (including its standard libraries’) static and dynamic guarantees, including but not limited to type and memory safety, which helps make our software both more secure and more safe

Memory Safety Is Hot

TLP:CLEAR

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

BACK TO THE BUILDING BLOCKS:

A Path Toward Secure and Measurable Software

FEBRUARY 2024

Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

"NSA advises organizations to [...] shift from [...] C/C++ [...] to a memory safe language when possible"

TLP:CLEAR

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation
Australian Signals Directorate's Australian Cyber Security Centre
Canadian Centre for Cyber Security
United Kingdom National Cyber Security Centre
New Zealand National Cyber Security Centre
Computer Emergency Response Team New Zealand

TLP:CLEAR

"C and C++ [...] can lead to memory unsafe code and are still among the most widely used languages today."

TLP:CLEAR

Product Security Bad Practices

Publication: October 2024

Cybersecurity and Infrastructure Security Agency

Federal Bureau of Investigation

This document is distributed as TLP:CLEAR. Recipients may distribute TLP:CLEAR information without restrictions. Information is subject to standard copyright rules. For more information on the Traffic Light Protocol, see cisa.gov/tlp.

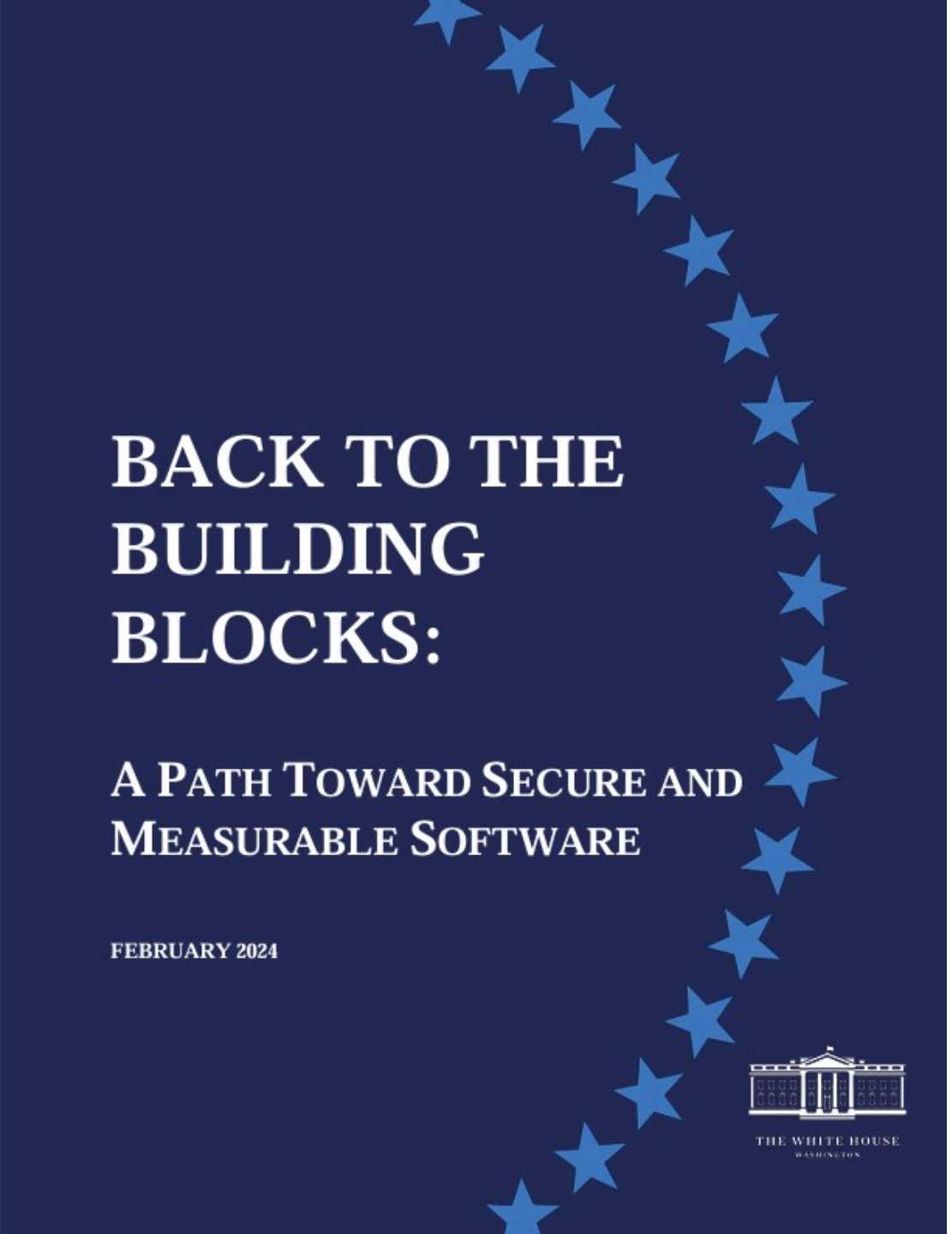
TLP:CLEAR

"[...] development of new product lines for use in service of **critical infrastructure or NCFs** in a memory-unsafe language (e.g., C or C++) where there are **readily available alternative memory-safe languages** that could be used is dangerous and significantly elevates risk to national security, national economic security, and national public health and safety."

Table 1-1: DIB Segments, Sub-segments, and Commodities

Industry Segments			
Industry Segments	**Industry Sub segment**	**Industry Segments**	**Industry Sub segment**
Aircraft	Fixed Wing	Munitions	Missile Tactical
	Rotary Wing		Missile Strategic
	Unmanned Aerial Systems		Missile Air/Air
Ships	Surface		Missile Air/Surface
	Sub-Surface		Missile Defense
	Unmanned Underwater Vehicles		Missile Surface/Air
Tracked and Wheeled Land Vehicles	Combat Vehicles		Missile Surface/Surface
	Tactical Vehicles		Precision Guided Munitions
	Unmanned Ground Vehicles		Ammunition
Electronics	Electronic Warfare		Missile Defense Agency
	Command, Control, Communications, Computer and Intelligence (C4I)	Space	Launch Vehicles
	Avionics		Satellite
Soldier Systems	Chemical Biological Defense Systems		Missile Defense Agency
	Clothing and Textiles	Mechanical	Transmissions (Air/Auto)
	Subsistence/Medical		Propulsion (Diesel/Rocket/ Turbine)
Structural	Castings/Forgings		Hydraulics
	Composites		Bearings
	Armor (Ceramic/Plating)		Nuclear Components (includes Depleted Uranium)
	Precious Metals		

https://www.cisa.gov/sites/default/files/publications/nipp-ssp-defense-industrial-base-2010-508.pdf



"**memory unsafe** *programming languages, such as* **C and C++**"

“Rust [...] has not yet been proven in space systems [...] Therefore, **to reduce memory safety vulnerabilities in space or other embedded systems** [...] **a complementary approach to implement memory safety** through hardware can be explored.”

C++ May Survive

- Gov’t memos admit there are fewer options for constrained and embedded systems
- It’s time and dollars expensive to switch
- C++ may yet prevail in the long run
  - See SafeCpp and other proposals

TL;DR

1)Review the Safety Profiles from the C++ Core Guidelines

2)Use clang-tidy to enforce C++ Core Guidelines Rules and subset the language

1) `clang-tidy main.cpp -checks=cppcoreguidelines-*`

3)Leverage the Guidelines Support Library to augment the language subset

Why is C++ Unsafe?

Type Safety

Bounds Safety

Lifetime Safety

Type Safety, Casting

```
// IEEE 754 representation of NaN
uint32_t nan_value = 0x7F800001;
float f = reinterpret_cast<float>(nan_value);
// May raise an exception
std::cout << f << std::endl;
```

Type Safety, Uninitialized Data

```
// Uninitialized data can have arbitrary values
int* p_data;

// Forget to initialize…

std::cout << “Data: ” << *p_data << std::endl;
```

Type Safety, Union

```
// We do this a lot with bits on HW systems
union Data {
 uint32_t i;
 float f;
}
Data data;
data.i = 0x7F800001;
// This is reserved value for float, NaN
std::cout << f;
```

Bounds Safety, Raw Array

```
// Raw arrays/pointers
char buffer[SIZE];
for(int i = 0; i=SIZE; i++){
 read(buffer[i]);  // WRONG: i = SIZE is out of bounds
}
```

Lifetime Safety, Use After Free

```
// lack of ownership/lifetime checking
int32_t* anInt = new int32_t(42);
delete anInt;
int32_t badRead = *anInt;
```

isocpp / CppCoreGuidelines

C++ Core Guidelines

February 15, 2024

Profiles summary:

- Pro.type: Type safety
- Pro.bounds: Bounds safety
- Pro.lifetime: Lifetime safety

GSL: Guidelines Support Library

GSL Public

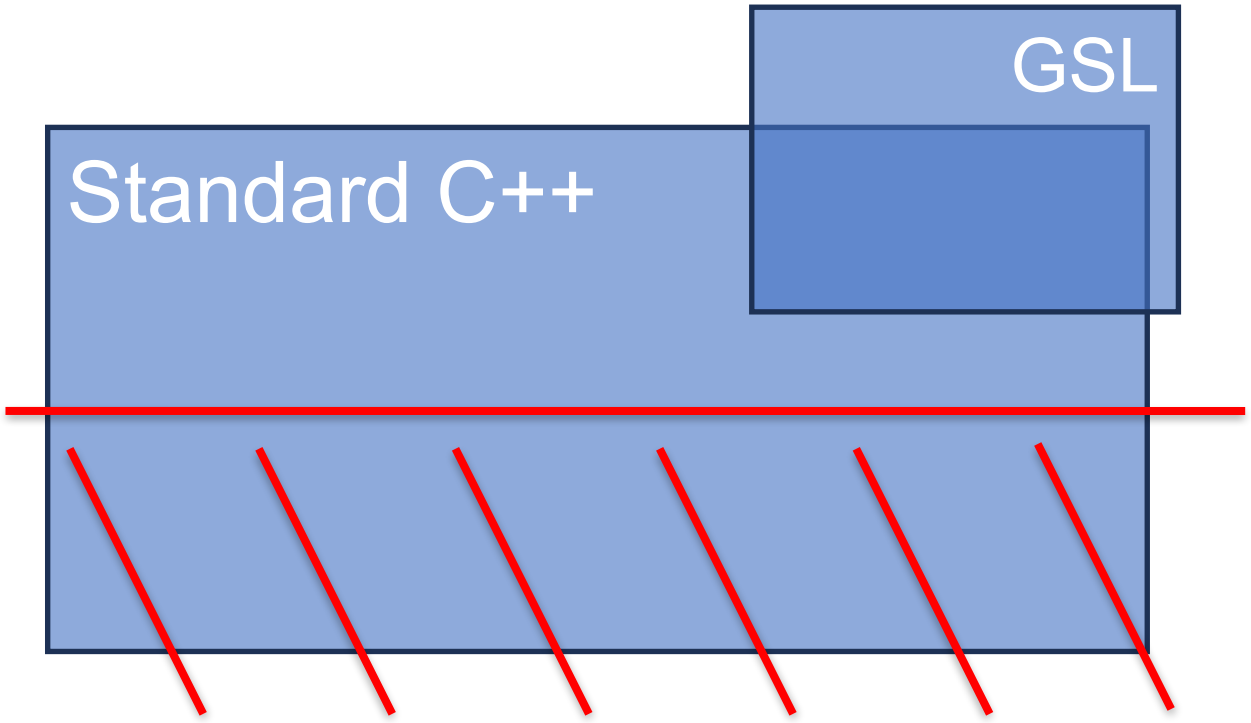
The Guidelines Support Library (GSL) contains functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation. This repo contains Microsoft's implementation of GSL. [...] The implementation **generally assumes** a platform that implements **C++14 support**.

gsl-lite Public

gsl-lite is a single-file header-only implementation of the C++ Core Guidelines Support Library originally based on Microsoft GSL and **adapted for C++98, C++03**. It also works when compiled as C++11, C++14, C++17, C++20.

Superset of a Subset

Do not use



Safety Profiles

- *A set of deterministic and portably enforceable rules designed to achieve specific guarantees.*
- *“Deterministic” means they need only local analysis and could be compiler-implemented.*
- *“Portably enforceable” means they function like language*

rules, providing consistent enforcement across tools.

- ***Conforming code is considered safe for targeted properties**,*

though errors may still arise from other code, libraries,

or external factors.

Type Safety Profile

[…] type-safety is defined to be **the property that a variable is not used in a way that doesn’t obey the rules for the type of its definition**. Memory accessed as a type T should not be valid memory that actually contains an object of an unrelated type U. Note that the safety is intended to be complete when combined also with Bounds safety and Lifetime safety.

Type Safety Profile Summary

Type.1: Avoid casts:

Don’t use reinterpret_cast; A strict version of Avoid casts and prefer named casts.

Don’t use static_cast for arithmetic types; A strict version of Avoid casts and prefer named casts.

Don’t cast between pointer types where the source type and the target type are the same; A strict version of Avoid casts.

Don’t cast between pointer types when the conversion could be implicit; A strict version of Avoid casts.

Type Safety Profile Summary

Type.2: Don’t use static_cast to downcast: Use dynamic_cast instead.

Type.3: Don’t use const_cast to cast away const (i.e., at all): Don’t cast away const.

Type.4: Don’t use C-style (T)expression or functional T(expression) casts: Prefer construction or named casts or T{expression}.

Type Safety Profile Summary

Type.5: Don’t use a variable before it has been initialized: always initialize.

Type.6: Always initialize a data member: always initialize, possibly using default constructors or default member initializers.

Type.7: Avoid naked union: Use variant instead.

Type.8: Avoid varargs: Don’t use va_arg arguments.

Bounds Safety Profile

We define bounds-safety to be **the property that a program does not use an object to access memory outside of the range that was allocated for it**. Bounds safety is intended to be complete only when combined with Type safety and Lifetime safety, which cover other unsafe operations that allow bounds violations.

Bounds Safety Profile Summary

Bounds.1: Don’t use pointer arithmetic. Use span instead: Pass pointers to single objects (only) and keep pointer arithmetic simple.

Bounds.2: Only index into arrays using constant expressions: Pass pointers to single objects (only) and Keep pointer arithmetic simple.

Bounds.3: No array-to-pointer decay: Pass pointers to single objects (only) and Keep pointer arithmetic simple.

Bounds.4: Don’t use standard-library functions and types that are not bounds-checked: Use the standard library in a type-safe manner.

Lifetime Safety

Accessing through a pointer that doesn’t point to anything is a major source of errors, and very hard to avoid in many traditional C or C++ styles of programming. For example, a pointer might be uninitialized, the nullptr, point beyond the range of an array, or to a deleted object.

Lifetime Safety Profile Summary

Lifetime.1: Don’t dereference a possibly invalid pointer: detect or avoid.

GSL Public

The Guidelines Support Library (GSL) contains functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation. This repo contains Microsoft's implementation of GSL. […] The implementation generally assumes a platform that implements C++14 support.

gsl-lite Public

gsl-lite is a single-file header-only implementation of the C++ Core Guidelines Support Library originally based on Microsoft GSL and adapted for C++98, C++03. It also works when compiled as C++11, C++14, C++17, C++20.

GSL Key Facilities

- `span<>`
- `variant<>`
- `owner<>`
- `unique_ptr<>`
- `shared_ptr<>`
- `not_null` and `nullptr`

Replace Raw Arrays With span<>

```
void pass_span(gsl::span<int> s)
{
 for(int i : s) { /* Range based for loop */ }
}
{
 int array[] = {1, 2, 3, 4, 5};
 gsl::span<int, 5> s(array);
 pass_span(s); // No array decay!
 s[6]; // Out of bounds error (customizable assert!)
}
```

Replace unions With variants<>

```
// Variants
std::variant<int, float> v;
v = 12.0f;
// Throws std::bad_variant_access
int i = std::get<int>(v);
// Avoid throw by checking for nullptr
const int* p_int = std::get_if<int>(&v);
// Valid pointer
const float* pf = std::get_if<float>(&v);
```

owner<>

```
// Zero overhead indicator that this pointer owns the object
void g(owner<int*> p, int* q, owner<int*> p2)
{
  // p = q; // bad: q is not an owner
  // delete q; // bad: q is not an owner
  q = p; // OK: q points to the object owned by p
  delete p; // needed: we are about to overwrite p
  p = p2; // OK: we just deleted p
  p2 = nullptr; // so that there are not two owners of *p2
  // *q = 7; // bad: assignment through now dangling pointer
  delete p; // needed: we are about to leave g()
}
```

Ownership Rules

- A pointer returned by new is an owner and must be deleted (unless stored in static storage to ensure that it lives “forever.”).
- Only a pointer known to be an owner can be deleted. Thus, a pointer passed into a scope as an owner<T*> must be deleted in that scope or passed along to another scope as an owner.
- A pointer that is passed into a scope as a plain T* may not be deleted.
- A pointer passed to another scope as an owner and not passed back as an owner is said to be invalidated and cannot be used again in its original scope (since it will have been deleted).

shared_ptr<>

```
{// Reference = 1
 std::shared_ptr<CoolThing> ptrToThing(new CoolThing());
{ // Reference count = 2
 std::shared_ptr<CoolThing> 2ndPtrToThing = ptrToThing;
 // Temporarily increases reference count
 passByValueFunction(2ndPtrToThing); // Reference count
drops back to 2 after function call
}
// 2ndPtrToThing is destroyed, reference count = 1
}
// ptrToThing is destroyed, reference count = 0,
// CoolThing is deleted
```

unique_ptr<>

```
{
 std::unique_ptr<MyObj> uniquePtr = std::make_unique<MyObj>();
 uniquePtr->doSomething();

 // Transfer ownership
 std::unique_ptr<MyObj> newOwnerPtr = std::move(uniquePtr);
 // Now, uniquePtr is null, and newOwnerPtr owns the object.
}
// The MyObj is automatically destroyed when newOwnerPtr
// goes out of scope.
```

Feature Replacement TL;DR

Drop	Adopt
C-Style Casts	Named Casts and brace-initialization
union {}	variant<>
raw_arrays[]	span<>
raw_pointers*	owner<> unique_ptr<> shared_ptr<>

Static Analysis

clang-tidy test.cpp -checks=-*,cppcoreguidelines-*

Core Guidelines Rule	Checker
Type.1 Avoid Casts: Don’t use reinterpret_cast	cppcoreguidelines-pro-type-reinterpret-cast
Type.1 Avoid Casts: Don’t use static_cast for arithmetic types	
Type.1 Avoid Casts: Don’t cast between pointer types that can be the same	
Type.1 Avoid Casts Don’t cast between pointer types that can be implict	
Type.2 Don’t use static to downcast, use dynamic_cast instead	cppcoreguidelines-pro-type-static-cast-downcast
Type.3 Don’t use const_cast to cast away const	cppcoreguidelines-pro-type-const-cast
Type.4: Don’t use C -Style casts	cppcoreguidelines-pro-type-cstyle-cast
Type.5: Don’t use a variable before it has been initialized	cppcoreguidelines-init-variables
Type.6: Always initialize a data member	cppcoreguidelines-pro-type-member-init
Type.7: Avoid naked union: Use variant instead.	cppcoreguidelines-pro-type-union-access
Type.8: Avoid varargs: Don’t use va_arg arguments.	cppcoreguidelines-pro-type-vararg
Bounds.1: Don’t use pointer arithmetic.	cppcoreguidelines-pro-bounds-pointer-arithmetic
Bounds.2: Only index into arrays using constant expressions:	cppcoreguidelines-pro-bounds-constant-array-index
Bounds.3: No array-to-pointer decay:	cppcoreguidelines-pro-bounds-array-to-pointer-decay
Bounds.4: Don’t use standard-library functions and types that are not bounds-checked	
Lifetime.1: Don’t dereference a possibly invalid pointer: detect or avoid.	cppcoreguidelines-owning-memory

Gaps

- Concurrency
- Iterator invalidation
- Aliasing

Safety Profiles In Context

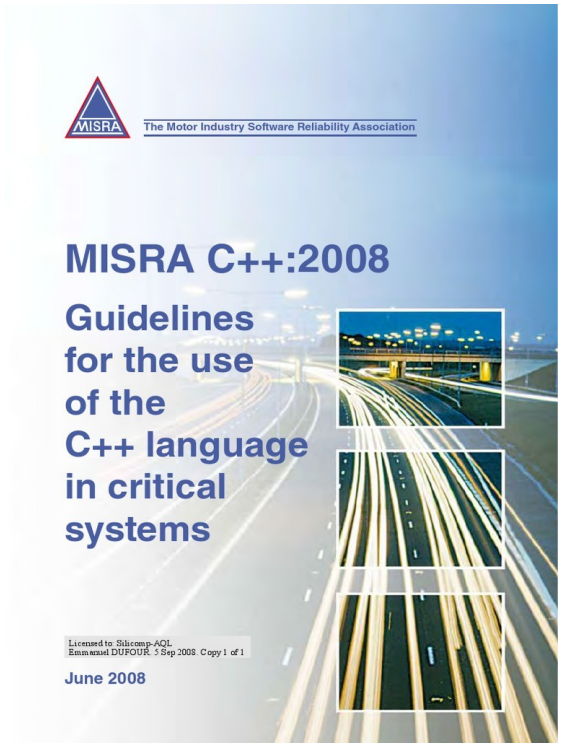
- Safety profiles represent the low-hanging fruit available today
- The safety guarantees are incomplete...
- New proposals with stronger guarantees are released frequently
- There is a push and pull to improve safety and avoid breaking syntax and ABI changes

References

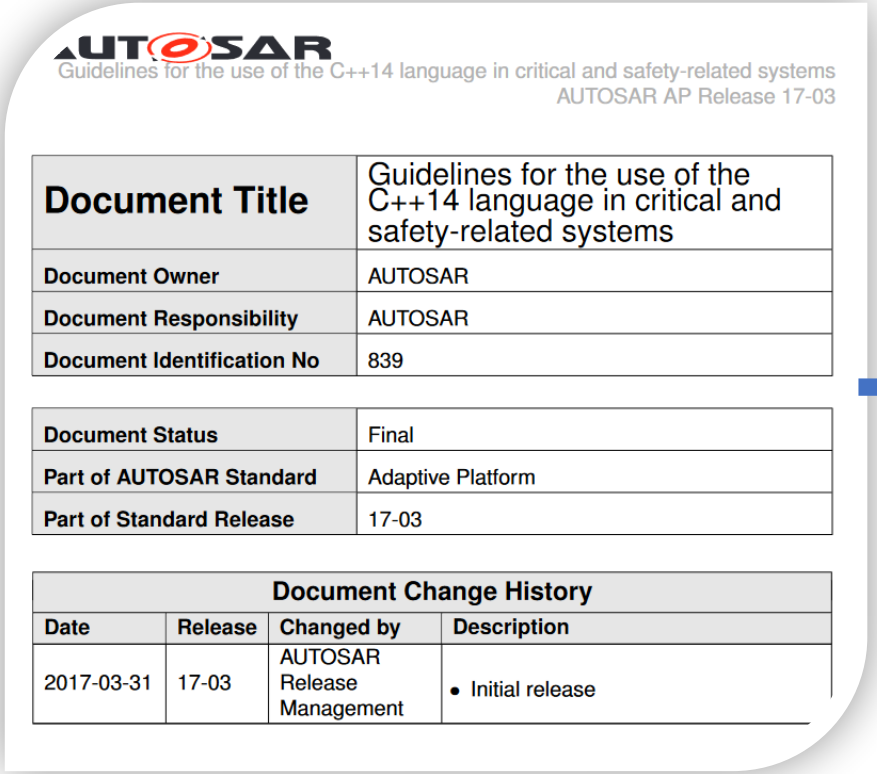
- github.com/isocpp/CppCoreGuidelines/docs
  - A brief introduction to C++'s model for type and resource safety
  - Lifetime Safety: Preventing Common dangling
- C++ Safety In Context, Herb Sutter
- Core Safety Profiles: Specification, adoptability, and impact
- SafeCpp Proposal

Backup

C++03



C++11/14


AUTOSAR

Guidelines for the use of the C++14 language in critical and safety-related systems
AUTOSAR AP Release 17-03

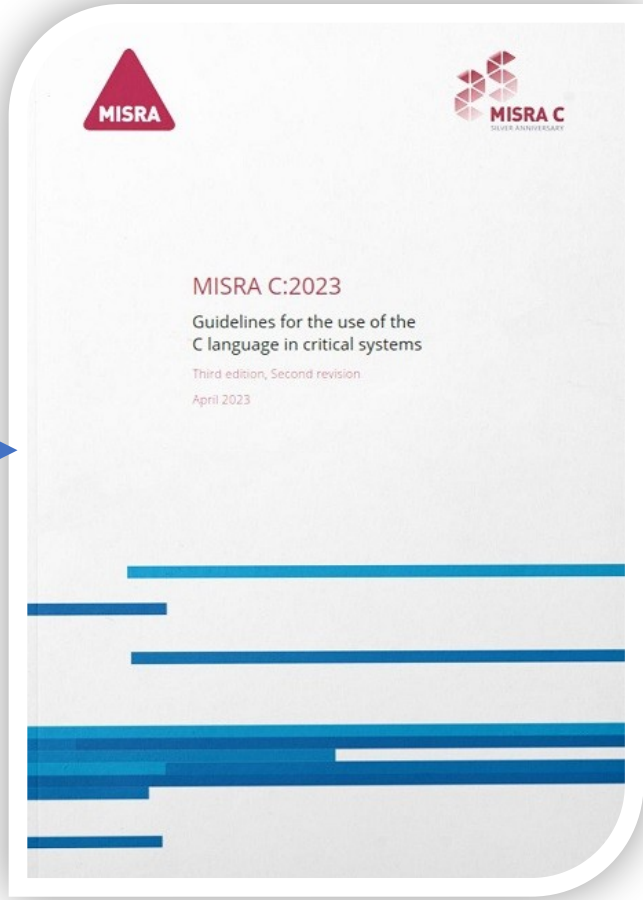
Document Title	Guidelines for the use of the C++14 language in critical and safety-related systems
Document Owner	AUTOSAR
Document Responsibility	AUTOSAR
Document Identification No	839

Document Status	Final
Part of AUTOSAR Standard	Adaptive Platform
Part of Standard Release	17-03

Document Change History			
Date	Release	Changed by	Description
2017-03-31	17-03	AUTOSAR Release Management	• Initial release

*

C++17



* AUTOSAR Press Release 1/29/2019 - MISRA will merge the AUTOSAR guidelines with its own established best practice to develop a single ‘go to’ language subset for safety-related C++ development. The MISRA led guidelines will incorporate the latest version of C++ language - C++17 - and, when available, its successor C++20.

National Security Agency | Cybersecurity Information Sheet

Software Memory Safety

Executive summary

Modern society relies heavily on software-based
developers to write software that operates in the
compromised for malicious purposes. While deve
prepare the logic in software for surprising condit
vulnerabilities are still frequently based on memo
overflowing a memory buffer and leveraging issu
allocates memory. Microsoft® revealed at a confe
70 percent of their vulnerabilities were due to me
found a similar percentage of memory safety vuln
Chrome®. [2] Malicious cyber actors can exploit t
execution or other adverse effects, which can oft
first step in large-scale network intrusions.

TLP:CLEAR

Australian Government
Australian Signals Directorate

ASD AUSTRALIAN SIGNALS DIRECTORATE
ACSC Australian Cyber Security Centre

Communications Security Establishment
Canadian Centre for Cyber Security

Centre de la sécurité des télécommunications
Centre canadien pour la cybersécurité

National Cyber Security Centre
a part of GCHQ

National Cyber Security Centre
PART OF THE GCSB

certnz

The Case for Memory Safe Roadmaps

Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously

Publication: December 2023

United States Cybersecurity and Infrastructure Security Agency
United States National Security Agency
United States Federal Bureau of Investigation

BACK TO THE BUILDING BLOCKS:

A Path Toward Secure and Measurable Software

FEBRUARY 2024

THE WHITE HOUSE
WASHINGTON

isocpp / CppCoreGuidelines

/docs

A brief introduction to C++'s model for type- and resource-safety

Bjarne Stroustrup (Morgan Stanley)
Herb Sutter (Microsoft)
Gabriel Dos Reis (Microsoft)

Lifetime safety: Preventing common dangling

Document Number: **P1179 R1 – version 1.1**
Date: 2019-11-22
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: Informational

C++ Core Guidelines

Profiles summary:

- Pro.type: Type safety
- Pro.bounds: Bounds safety
- Pro.lifetime: Lifetime safety

GSL: Guidelines Support Library

GSL Public

The Guidelines Support Library (GSL) contains functions and types that are suggested for use by the C++ Core Guidelines maintained by the Standard C++ Foundation. This repo contains Microsoft's implementation of GSL. [...] The implementation **generally assumes** a platform that implements **C++14 support**.

gsl-lite Public

gsl-lite is a single-file header-only implementation of the C++ Core Guidelines Support Library originally based on Microsoft GSL and **adapted for C++98, C++03**. It also works when compiled as C++11, C++14, C++17, C++20.

isocpp / CppCoreGuidelines

A brief introduction to C++'s model for type- and resource-safety

Bjarne Stroustrup (Morgan Stanley)
Herb Sutter (Microsoft)
Gabriel Dos Reis (Microsoft)

Lifetime safety: Preventing common dangling

Document Number: **P1179 R1 – version 1.1**
Date: 2019-11-22
Reply-to: Herb Sutter (hsutter@microsoft.com)
Audience: Informational

C++ Core Guidelines

February 15, 2024

Profiles summary:

- Pro.type: Type safety
- Pro.bounds: Bounds safety
- Pro.lifetime: Lifetime safety