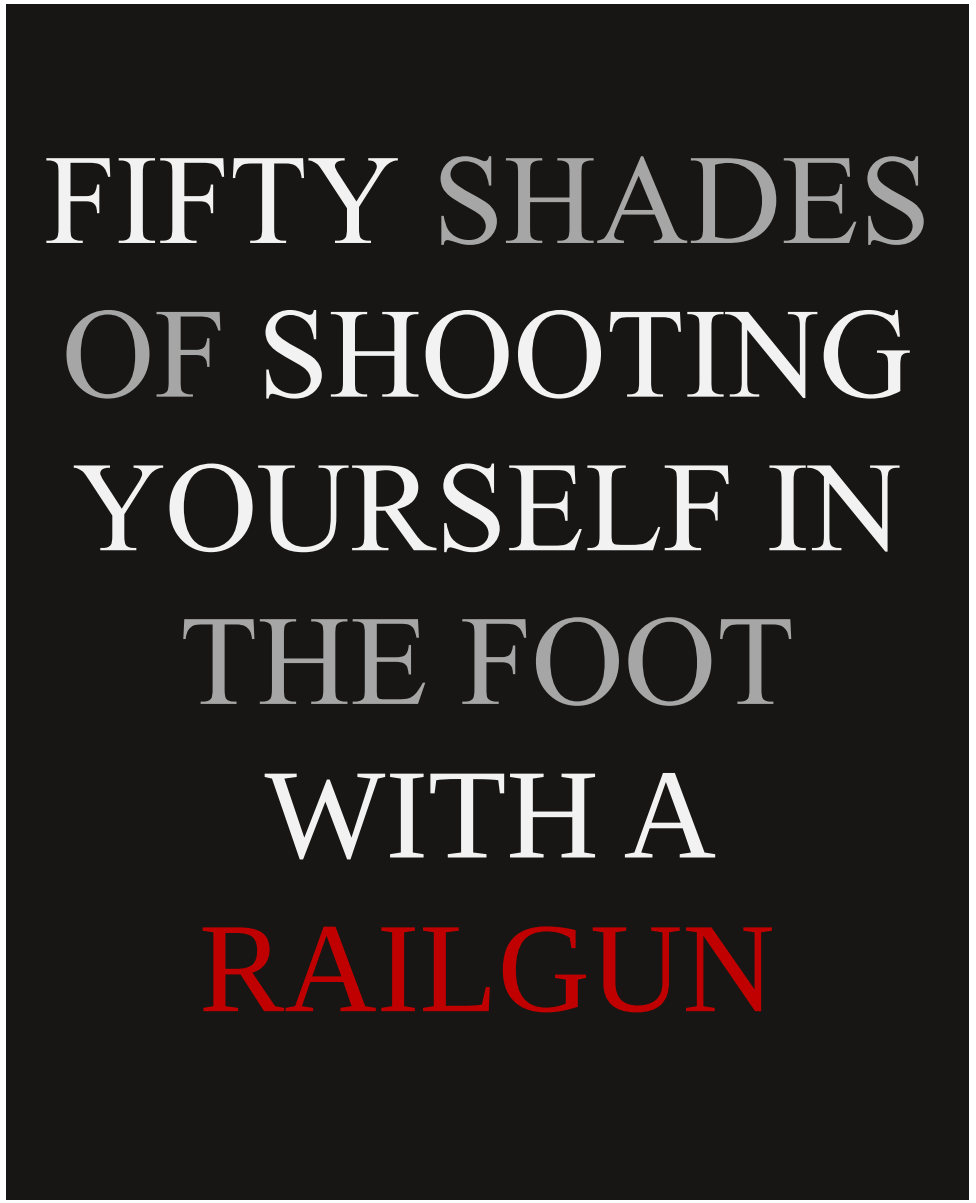
Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design

MAX HOFFMANN

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design


FIFTY SHADES
OF SHOOTING
YOURSELF IN
THE FOOT
WITH A
RAILGUN


THE WHITE HOUSE
MENU
FEBRUARY 26, 2024
Press Release: Future Software Should Be Memory Safe
ONCD
BRIEFING ROOM
PRESS RELEASE
Leaders in Industry Support White House Call to Address Root Cause of Many of the Worst Cyber Attacks

Common Weakness Enumeration

A Community-Developed List of Software & Hardware Weakness Types

Table 1. Stubborn Weaknesses in the CWE Top 25

CWE-ID	Description		
CWE-787	Out-of-bounds Write		
CWE-79	Improper Neutralization of Input During Web Page Gene… Scripting')		
CWE-89	Improper Neutralization of Special Elements used in an SQ… ('SQL Injection')		
CWE-416	Use After Free		
CWE-78	Improper Neutralization of Special Elements used in an OS… Command Injection')		
CWE-20	Improper Input Validation	View	6
CWE-190	Integer Overflow or Wraparound	View	14
CWE-502	Deserialization of Untrusted Data	View	15
CWE-119	Improper Restriction of Operations within Bounds of a Memory Buffer	View	17

15 weaknesses that have been present in **every** CWE Top 25 of the past 5 years

Weaknesses associated with languages that do not have strong support for memory management or type enforcement.
CWE-119, CWE-190, CWE-416, CWE-787, and CWE-476 are in this group. Of these, CWE-787, CWE-416, and CWE-352 were ranked in the top 10 for all five years, while CWE-476, CWE-287, and CWE-798 were mainstays in positions 12-20.

https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html

BLEEPINGCOMPUTER

Home > News > Microsoft > Zero-click Windows TCP/IP RCE impacts all systems with IPv6 enabled, patch now

Zero-click Windows TCP/IP RCE impacts all systems with IPv6 enabled, patch now

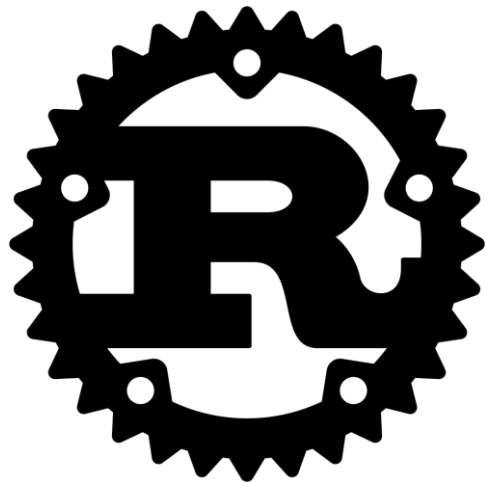
By **Sergiu Gatlan**

August 14, 2024 04:51 PM 0

Found by Kunlun Lab's XiaoWei and tracked as CVE-2024-38063, this security bug is caused by an Integer Underflow weakness, which attackers could exploit to trigger buffer overflows that can be used to execute arbitrary code on vulnerable Windows 10, Windows 11, and Windows Server systems.

Rust

A language empowering everyone
to build reliable and efficient software.



C++ Core Guidelines

May 11, 2024

Editors:

- Bjarne Stroustrup
- Herb Sutter

Smart pointers

Smart pointers enable automatic, exception-safe, object lifetime management.

Defined in header <memory>

Pointer categories

unique_ptr (C++11)	smart pointer with unique object ownership semantics (class template)
shared_ptr (C++11)	smart pointer with shared object ownership semantics (class template)

Standard library header <ranges> (C++20)

This header is part of the ranges library.

Namespace aliases

```
namespace std {
    namespace views = ranges::views;
}
```

The namespace alias `std::views` is provided as a shorthand for `std::ranges::views`.

ISO/IEC JTC1 SC22 WG21 P2795R5

Date: 2024-03-22

To: SG12, SG23, EWG, CWG, LWG

Thomas Köppe <tkoeppe@google.com>

Erroneous behaviour for uninitialized reads

cppfront

Copyright (c) Herb Sutter • See License

```
main: () = std::cout << "Hello, world!\n";
```

Contributor Covenant 2.1 | Multi-platform Build of cppfront passing

Cppfront is a compiler from an experimental C++ 'syntax 2' (Cpp2) to today's 'syntax 1' (Cpp1), to prove out some concepts, share some ideas, and prototype features that can also be proposed for evolving today's C++.

Carbon Language:
An experimental successor to C++

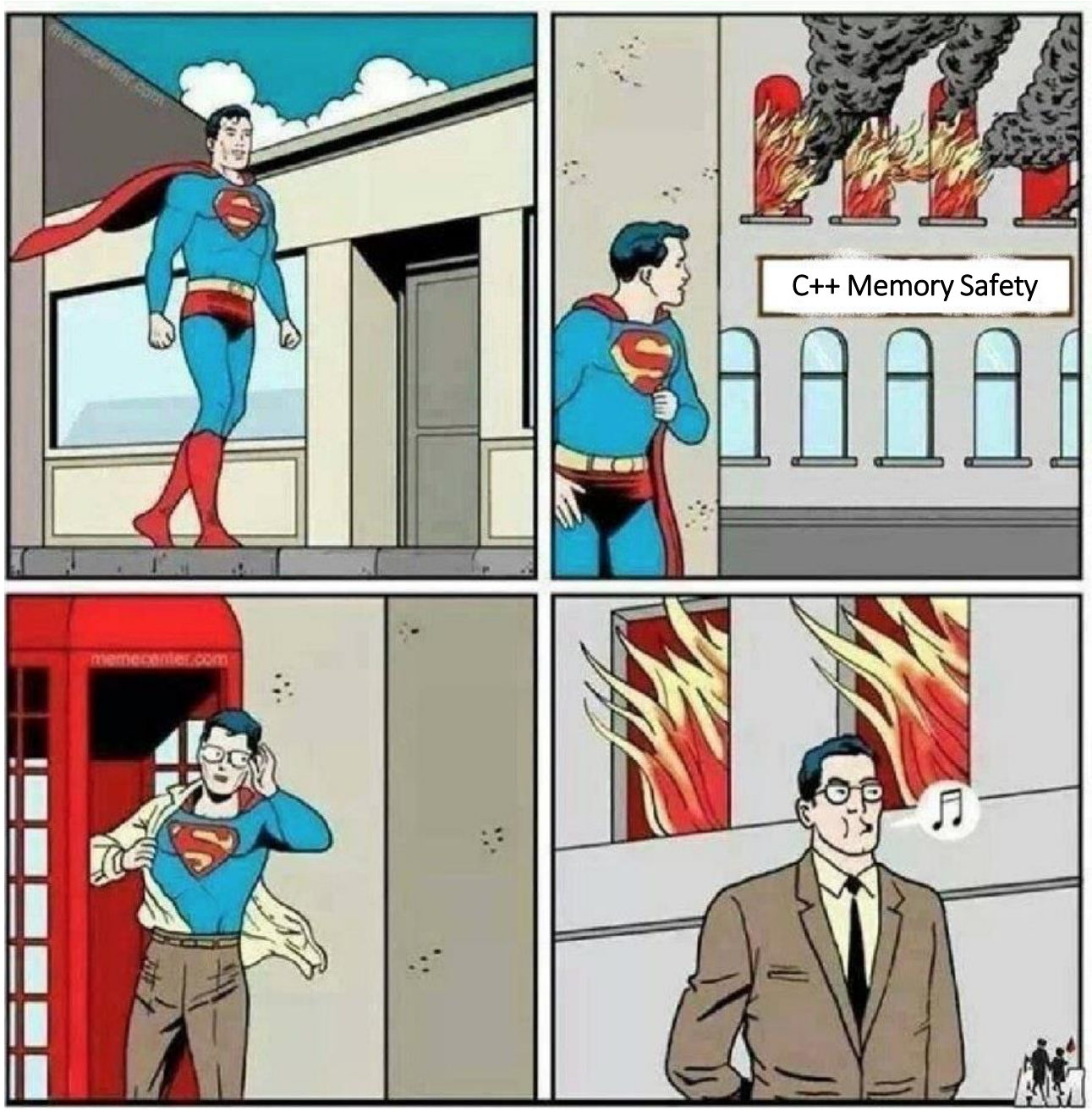
Memory safety

Safety, and especially memory safety, remains a key challenge for C++ and something a successor language needs to address. Our initial priority and focus is on immediately addressing important, low-hanging fruit in the safety space:

New Circle is out!

- New Circle notes

New Circle is a major transformation of the Circle compiler, intended as a response to recent successor language announcements. It focuses on a novel fine-grained versioning mechanism that allows the compiler to **fix defects** and make the language **safer** and **more productive** while maintaining **100% compatibility** with existing code assets.


C++ Memory Safety

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design

ANDY GREENBERG SECURITY JUL 24, 2015 12:30 PM

After Jeep Hack, Chrysler Recalls 1.4M Vehicles for Bug Fix

Welcome to the age of hackable automobiles, when two security researchers can cause a 1.4 million product recall.

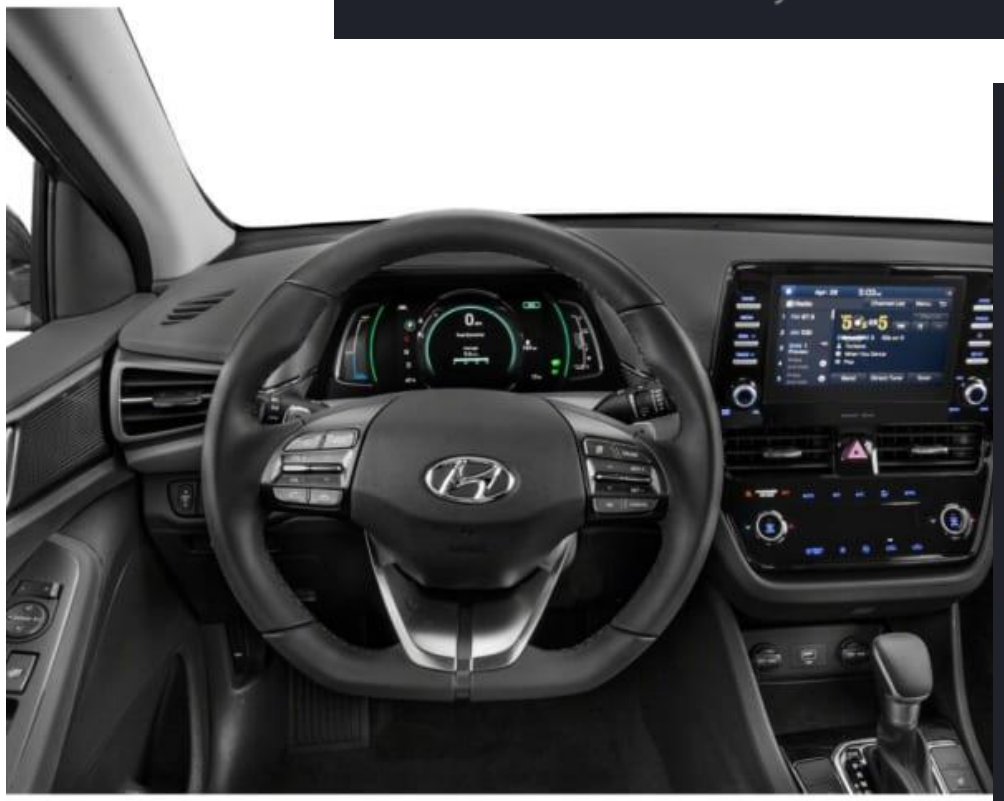
ANDY GREENBERG/WIRED

https://www.wired.com/2015/07/jeep-hack-chrysler-recalls-1-4m-vehicles-bug-fix/

How I Hacked my Car

2022-05-22 :: greenluigi1

#d-audio #d-audio2 #hyundai #kia #hacking #car #IVI #howIHackedMyCar



It is… Beautiful

After launching the new binary I was greeted with the beautiful sight of Doom running in my car.

https://programmingwithstyle.com/posts/howihackedmycar/

Dr.-Ing. Max Hoffmann

Ph.D. at the research group **Embedded Security** of Prof. Dr.-Ing. Christof Paar @ RUB &

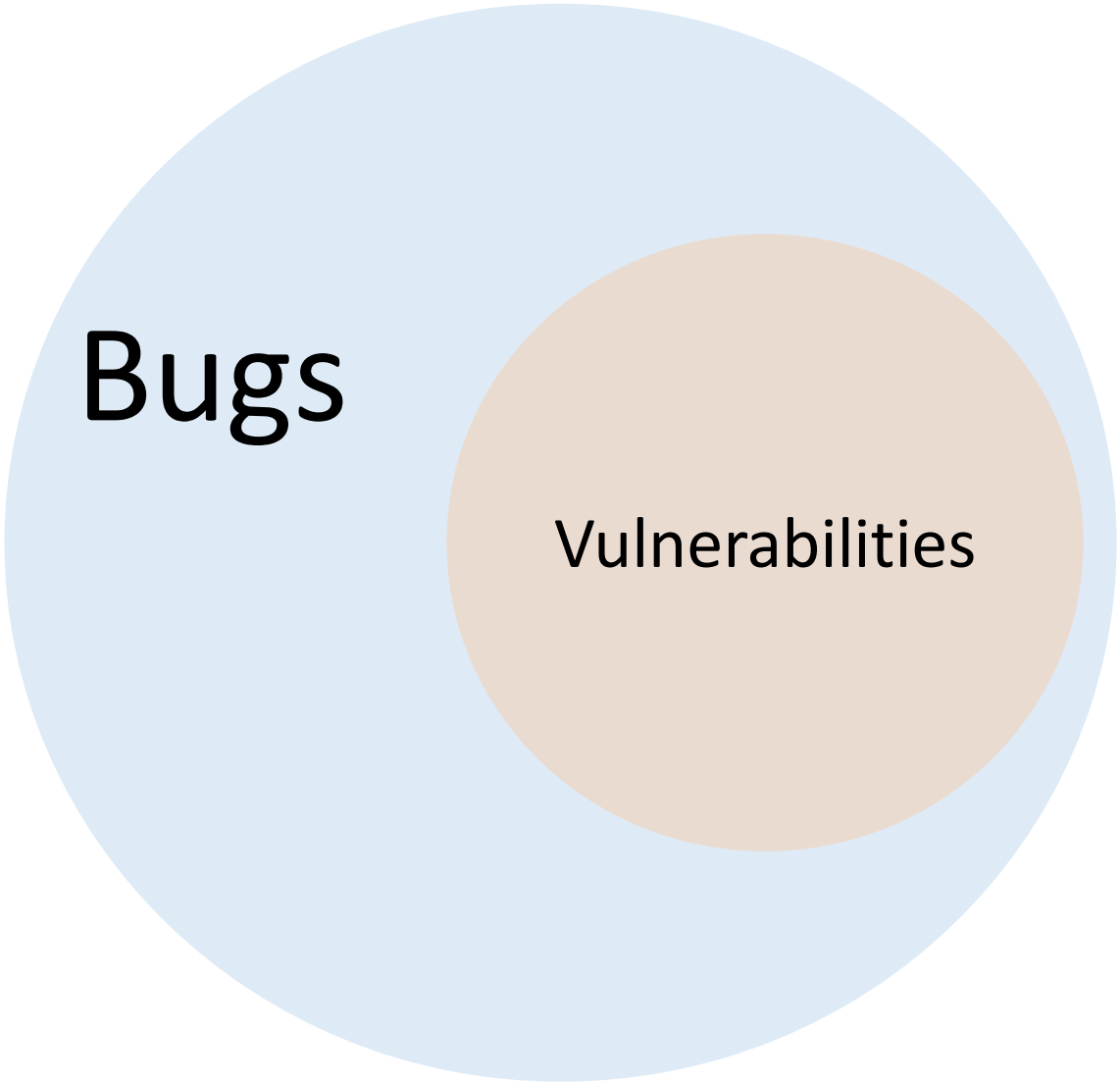
Product Field **Security Manager** @ ETAS

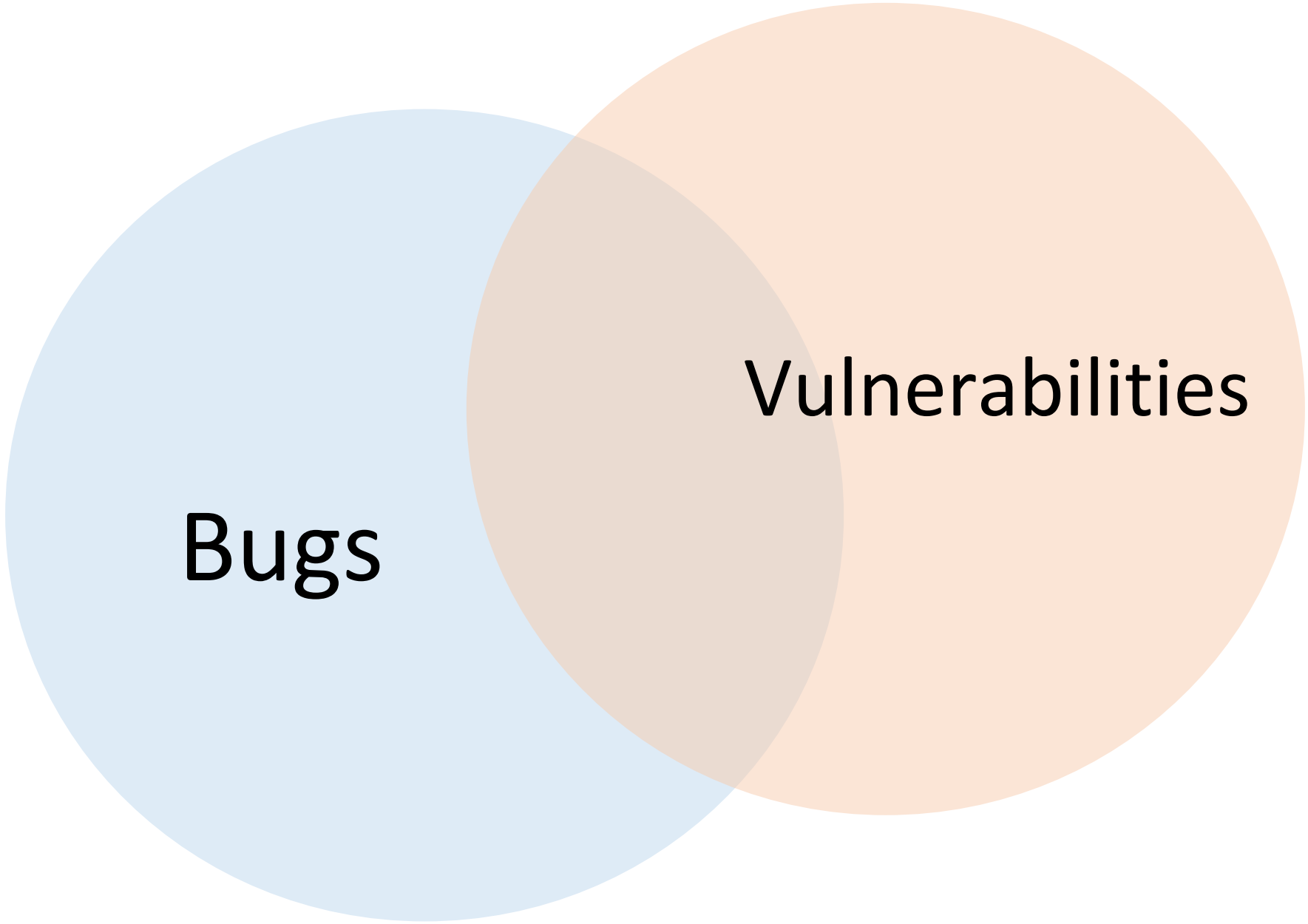
External Lecturer @ RUB

Views expressed are my own and not necessarily reflect those of my employer.

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design


Bugs
Vulnerabilities


Bugs
Vulnerabilities

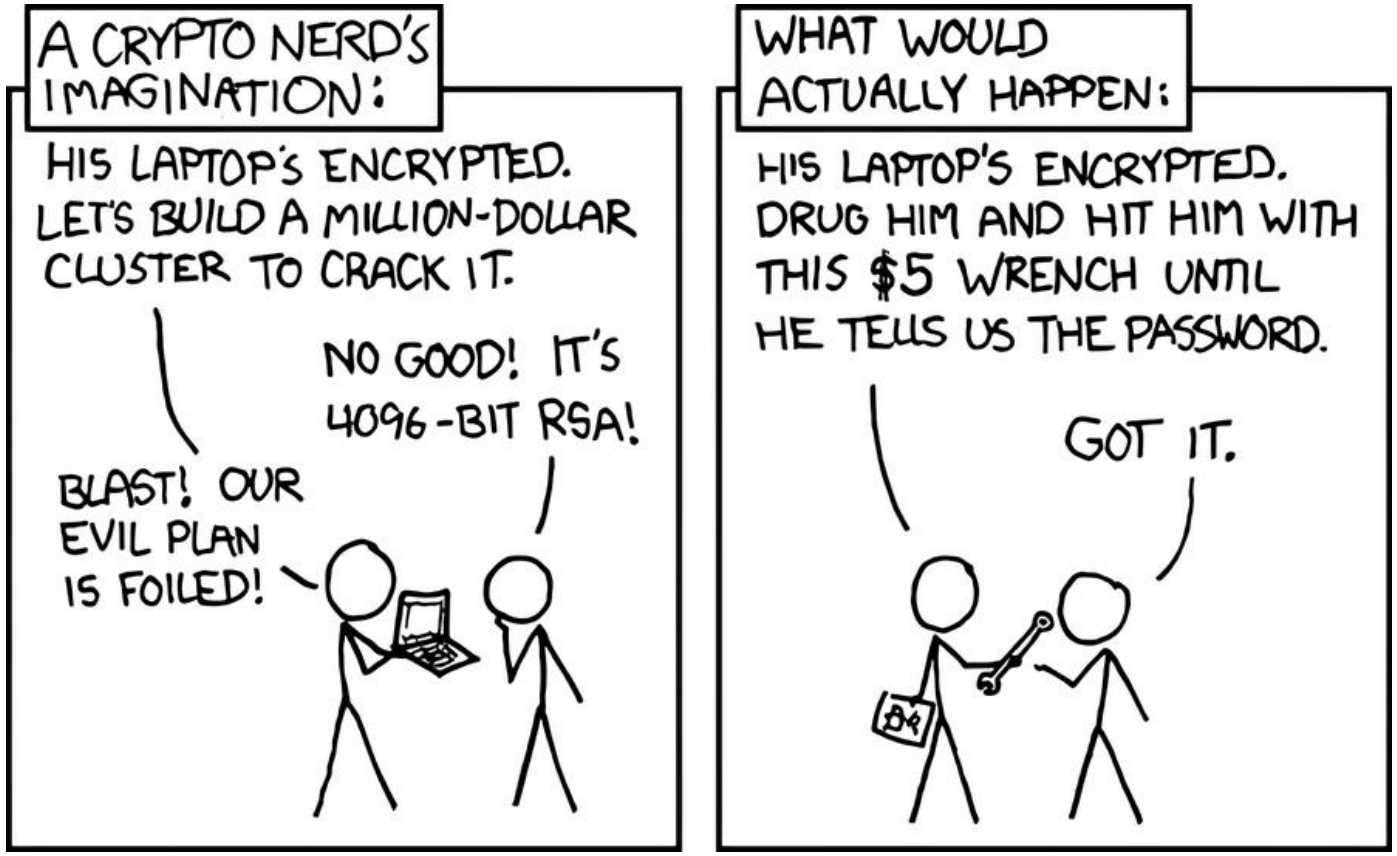
Security is not about
what is intended,
but about
what is possible.

Patch Notes v3.5.61

- Increased parking ticket price
- Prevented authentication bypass at entry 42
- …

Google Street View: University Bielefeld, Germany

Arguing about security is mostly meaningless without a proper **attacker model**.

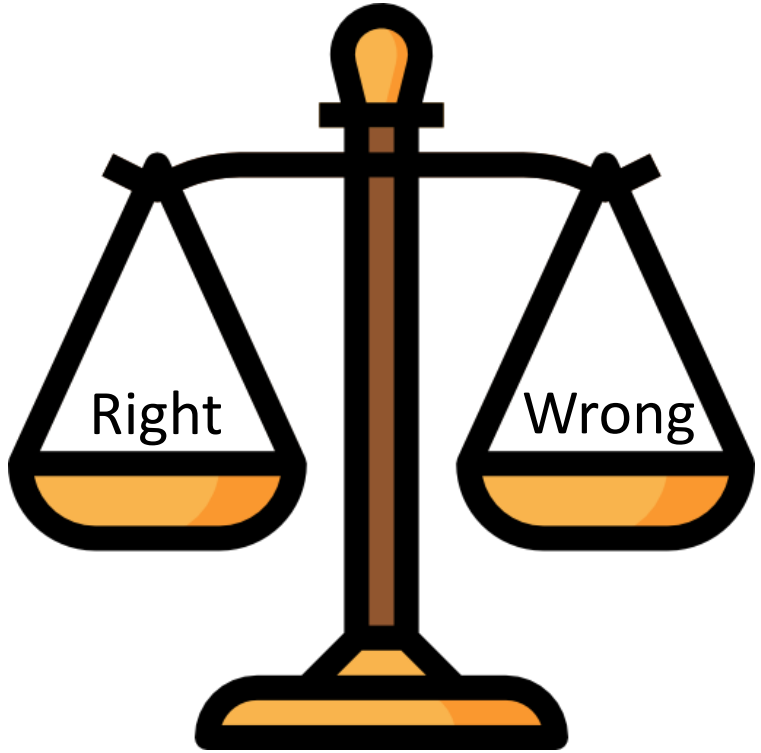


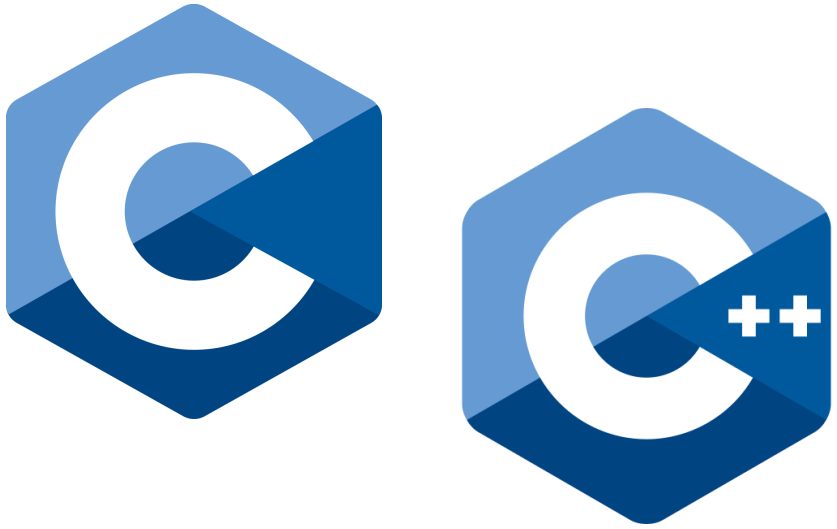
https://xkcd.com/538/

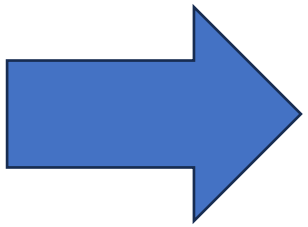
Security is inherently **complex**. A single mistake can be devastating.

Solutions should be **simple** to reduce the risk of mistakes.

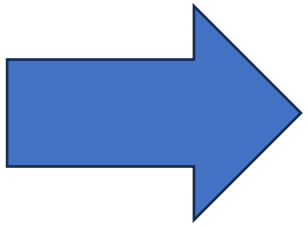
Make the **right thing easy** and the **wrong thing hard** to do.







Trust the Developer

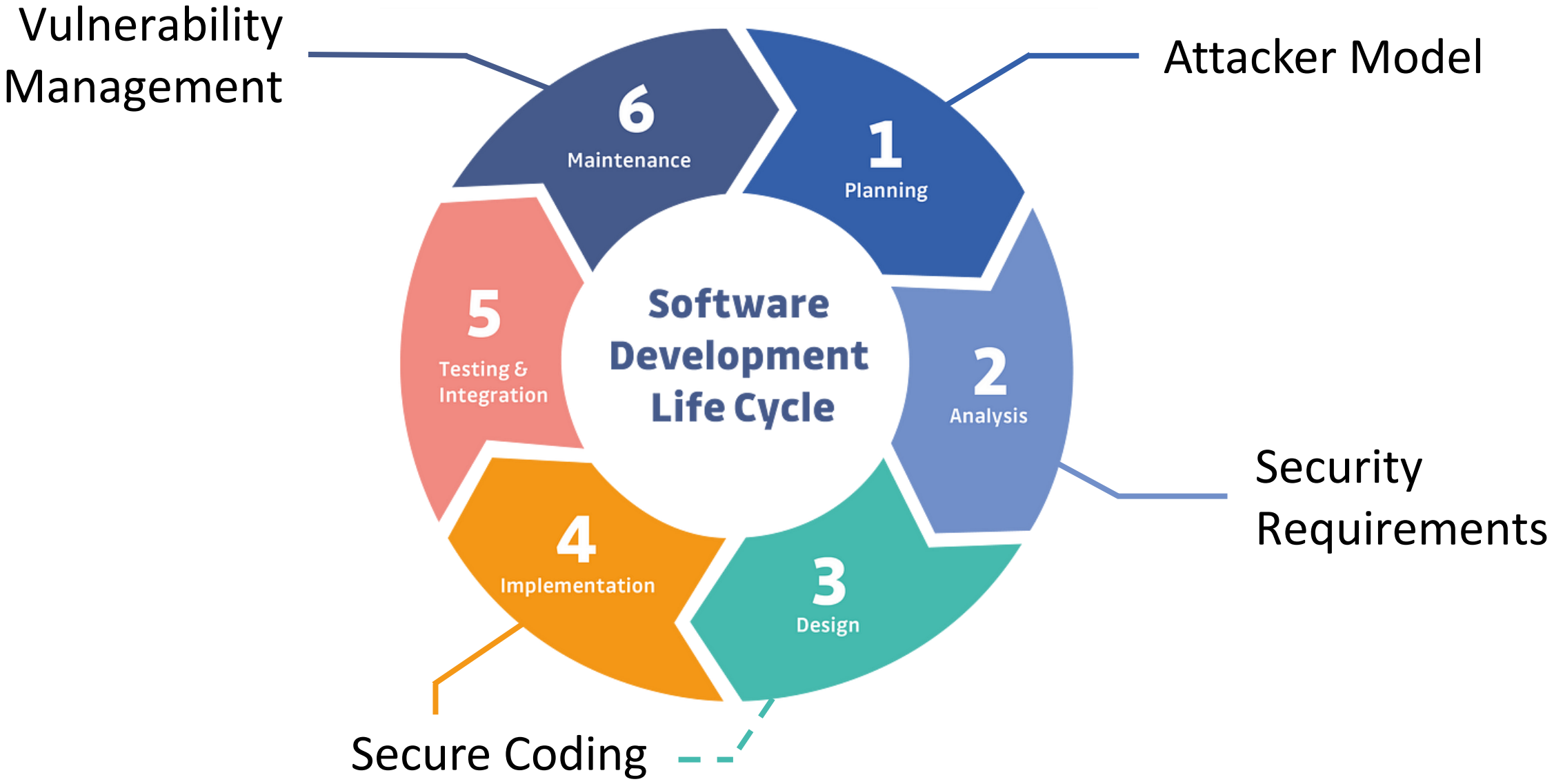


Trust the Compiler*

* a little more

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design


Vulnerability Management
Attacker Model
6
Maintenance
1
Planning
5
Testing & Integration
Software Development Life Cycle
2
Analysis
Security Requirements
4
Implementation
3
Design
Secure Coding

- Setting Expectations -

My Bias: Embedded Security

Goals:

- Show examples that
  - guard against accidental mistakes
  - reduce cognitive load in reviews
- Provide inspirations for your use cases

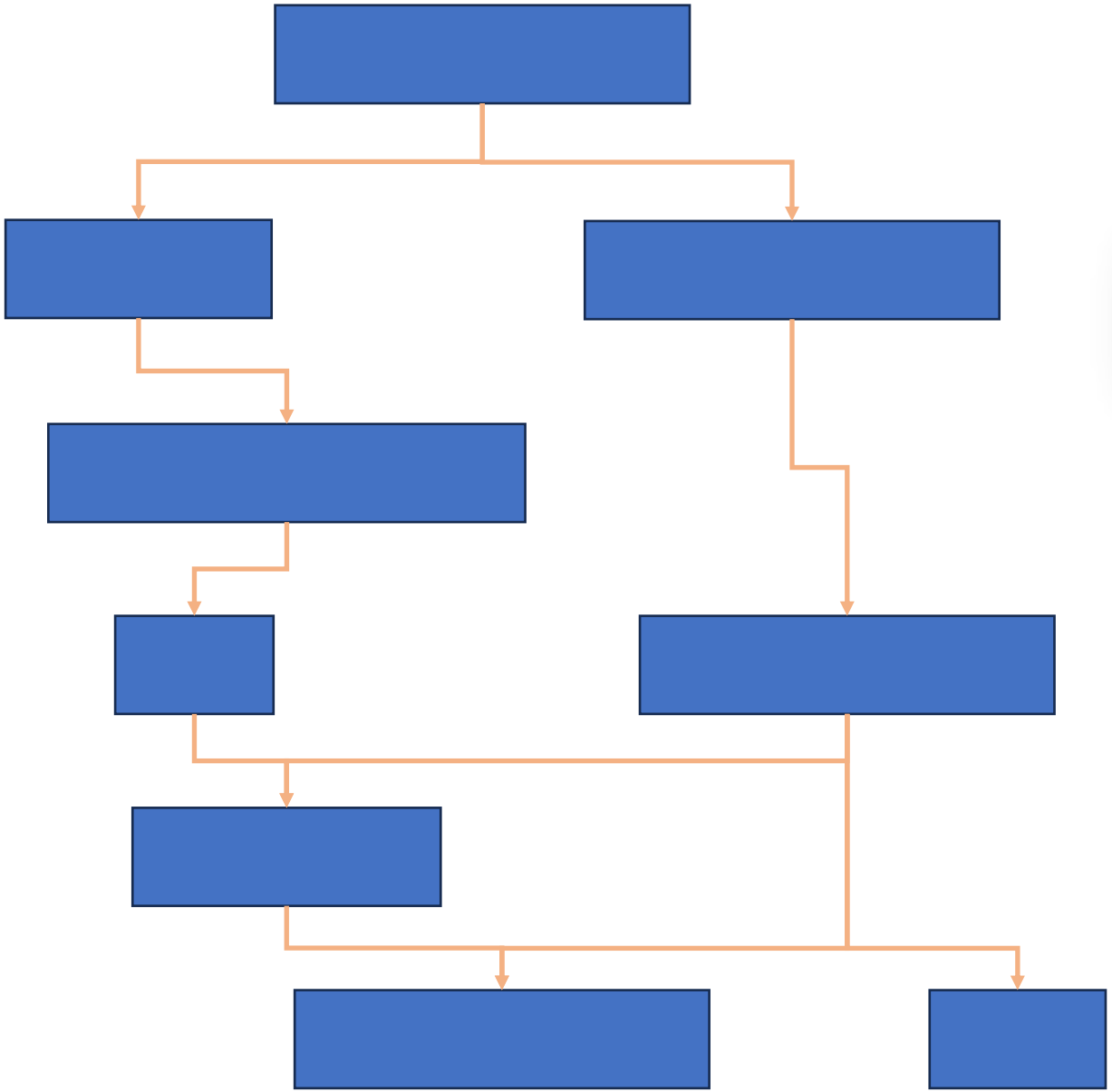
Non-Goals:

- Show production-ready code
- Present this as the only solution
- Present a “plug’n’play” solution


SECRET

```
1  std::array<char, 10> password_buffer;
2  get_password(password_buffer);
3  bool valid = try_authenticate(password_buffer);
4  std::memset(password_buffer, 0, sizeof(password_buffer));
```

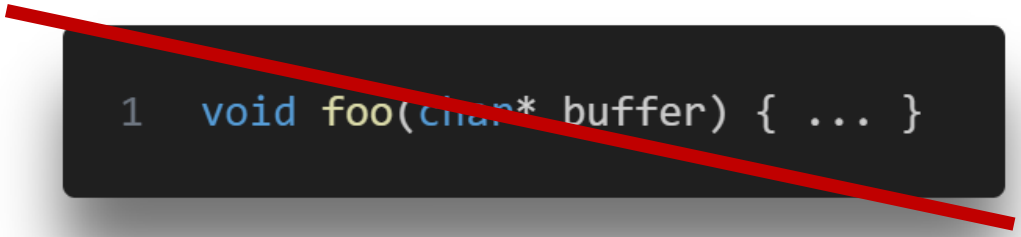
```
1  std::array<char, 10> password_buffer;
2  get_password(password_buffer);
3  bool valid = try_authenticate(password_buffer);
4  secure_erase_bytes(password_buffer, sizeof(password_buffer));
```



```
1   std::array<char, 10> password_buffer;
```

```
1   void foo(char* buffer) { ... }
```

```cpp
template<typename T>
class Sensitive
{
    T m_content;

public:
    Sensitive() = default;

    void secure_erase();

    ~Sensitive() { secure_erase(); }

    // remove copy operations

    Sensitive(Sensitive&& other)
    {
        m_content = other.m_content;
        other.secure_erase();
    }
    // + add move assignment operator

    using Func = void (*)(T&);
    void with_sensitive_content(Func&& func) { func(m_content); }
};
```



Either…

```
1  void foo(Sensitive<...>& buffer) { ... }
```

"Ah `buffer` is somewhat secret! Better be careful."

…Or

```
1  x.with_sensitive_content(foo);
```

Screams "**REVIEW ME THOROUGHLY**" during code reviews

```cpp
template<typename T>
class Sensitive
{
    T m_content;

public:
    Sensitive() = default;

    void secure_erase();

    ~Sensitive() { secure_erase(); }

    // remove copy operations

    Sensitive(Sensitive&& other)
    {
        m_content = other.m_content;
        other.secure_erase();
    }
    // + add move assignment operator

    using Func = void (*)(T&);
    void with_sensitive_content(Func&& func) { func(m_content); }
};
```

```
void *memset( void *dest, int ch, size_t count );                          (1)
void *memset_explicit( void *dest, int ch, size_t count );                 (2)   (since C23)
errno_t memset_s( void *dest, rsize_t destsz, int ch, rsize_t count );     (3)   (since C11)
```

Both still largely unsupported

Tested in Compiler Explorer

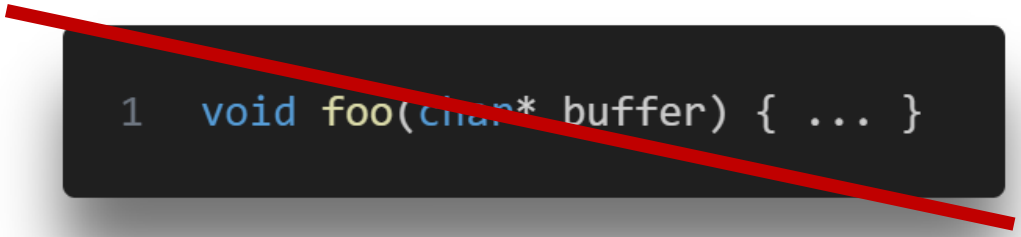
IACR Transactions on Cryptographic Hardware and Embedded Systems
ISSN 2569-2925, Vol. 2024, No. 1, pp. 375–397. DOI:10.46586/tches.v2024.i1.375-397

High-assurance zeroization

Santiago Arranz Olmos[1], Gilles Barthe[1,2], Ruben Gonzalez[1,6], Benjamin Grégoire[3], Vincent Laporte[4], Jean-Christophe Léchenet[3], Tiago Oliveira[1] and Peter Schwabe[1,5]

[1] MPI-SP, Bochum, Germany
[2] IMDEA Software Institute, Madrid, Spain
[3] Inria, Sophia-Antipolis, France
[4] Inria, Nancy, France
[5] Radboud University, Nijmegen, The Netherlands
[6] Neodyme AG, Munich, Germany

Abstract. In this paper we revisit the problem of erasing sensitive data from memory and registers during return from a cryptographic routine. While the problem and related attacker model is fairly easy to phrase, it turns out to be surprisingly hard to guarantee security in this model when implementing cryptography in common languages such as C/C++ or Rust. We revisit the issues surrounding zeroization and then present a principled solution in the sense that it guarantees that sensitive data



Either…

```
1  void foo(Sensitive<...>& buffer) { ... }
```

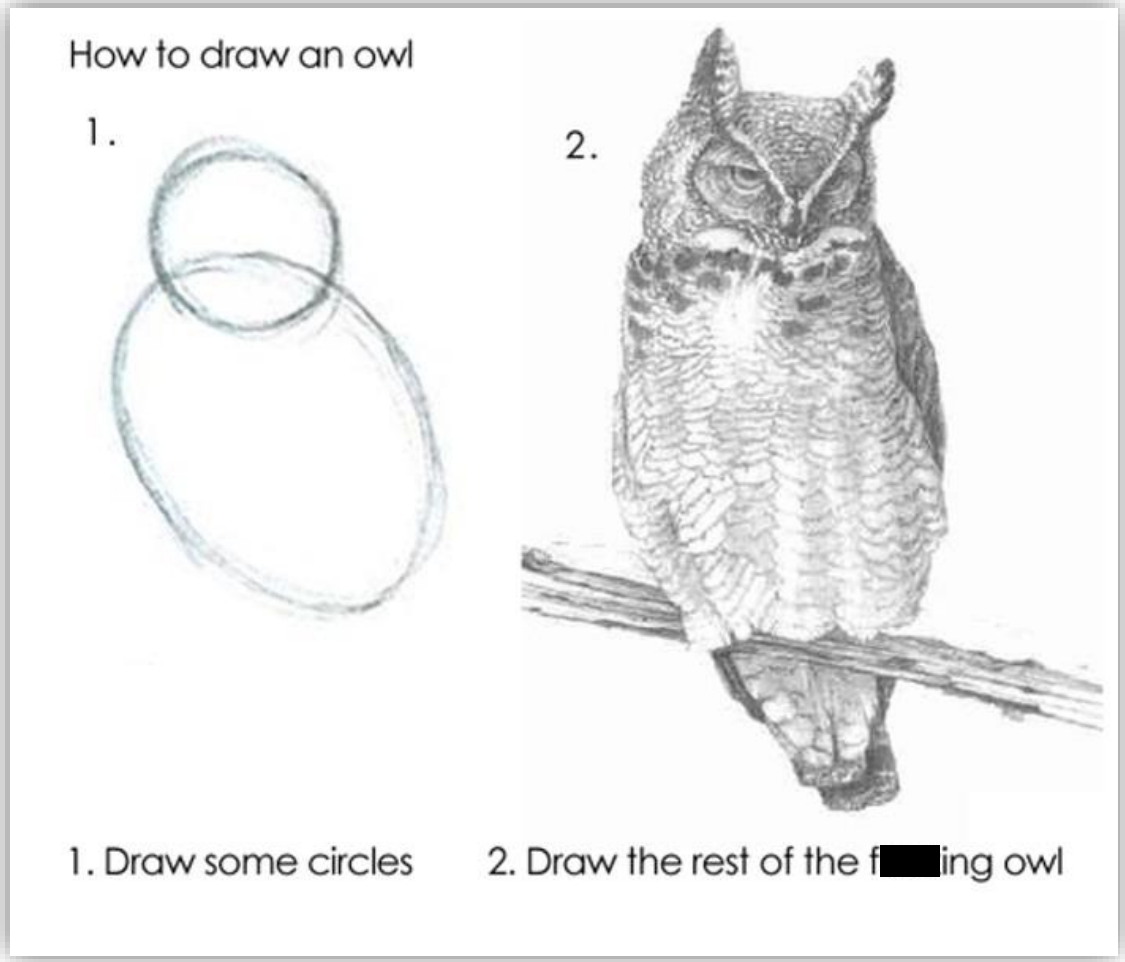
"Ah `buffer` is somewhat secret! Better be careful."

…Or

```
1  x.with_sensitive_content(foo);
```

Screams "**REVIEW ME THOROUGHLY**" during code reviews

Tutorial Completed



Our Product

"C++ rocks!"

Message Receiver Application
(written by customer)

Update Text Command: "C++ rocks!"

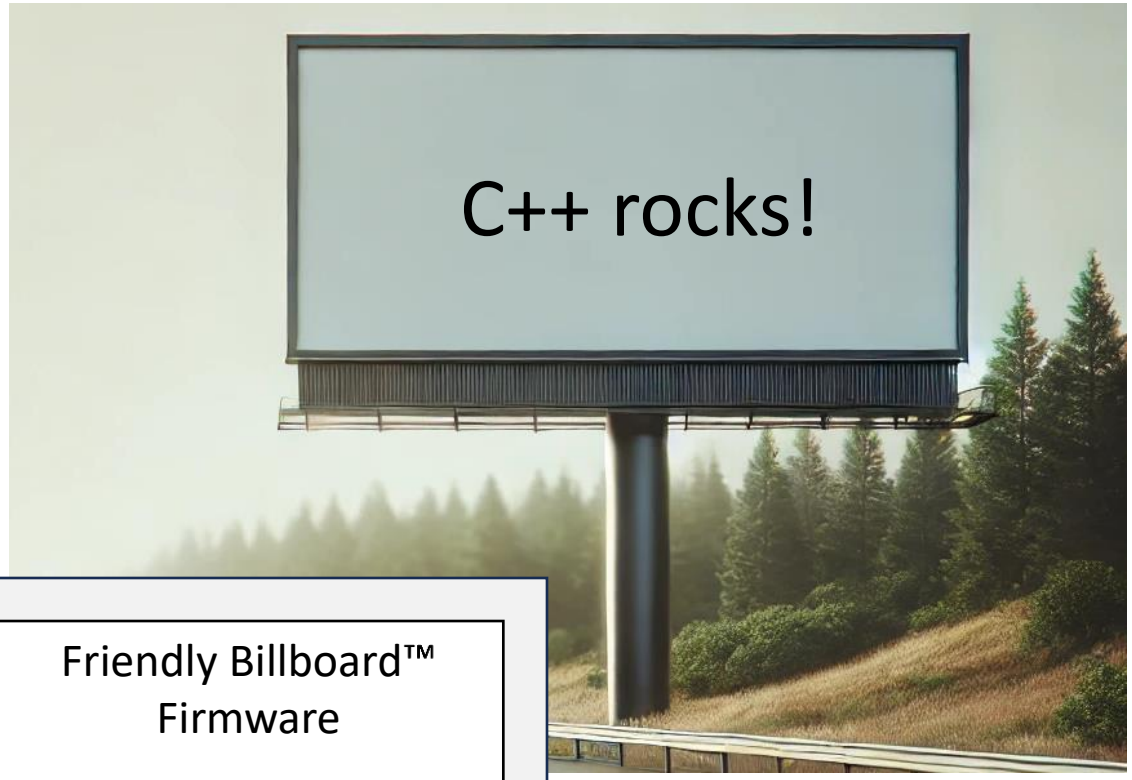
Shared Memory

Friendly Billboard™ Firmware
(written by us)

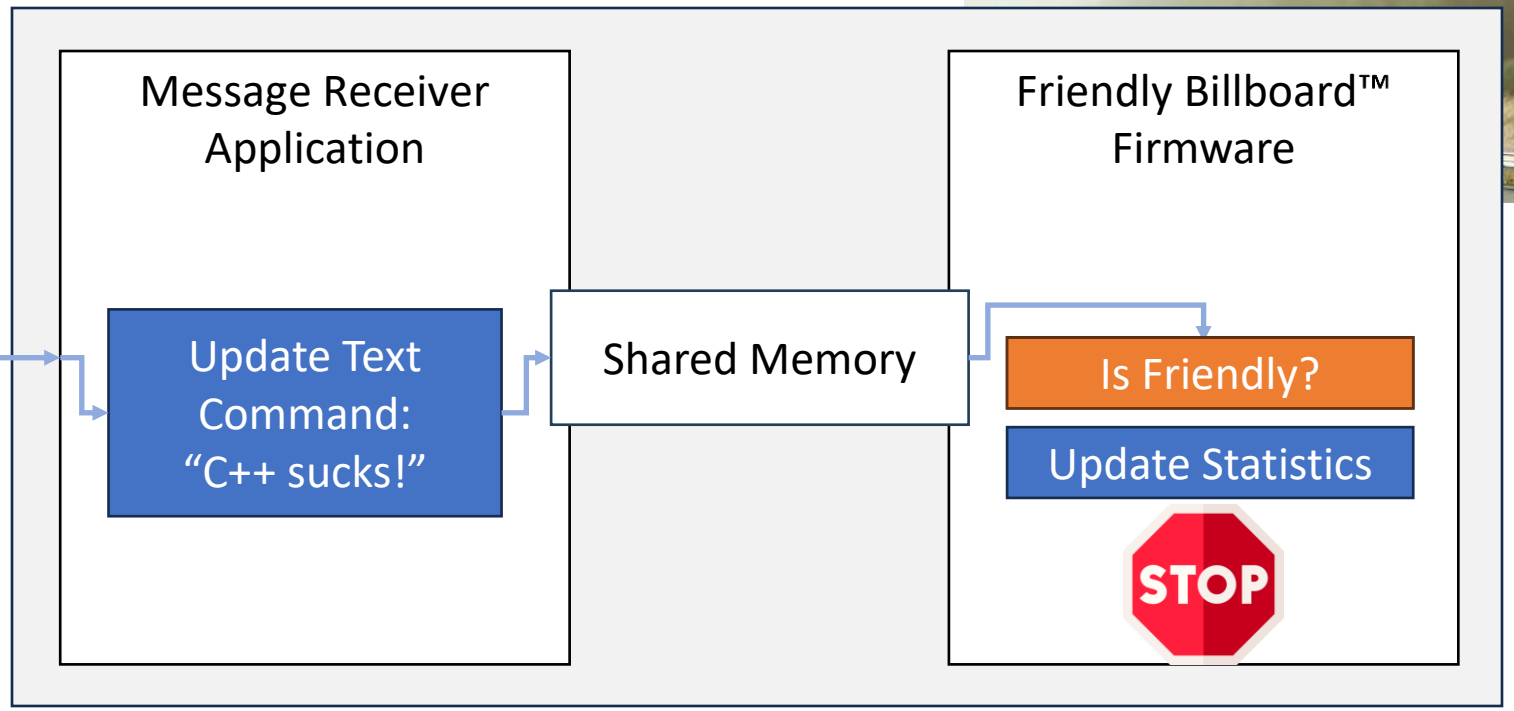
Is Friendly?

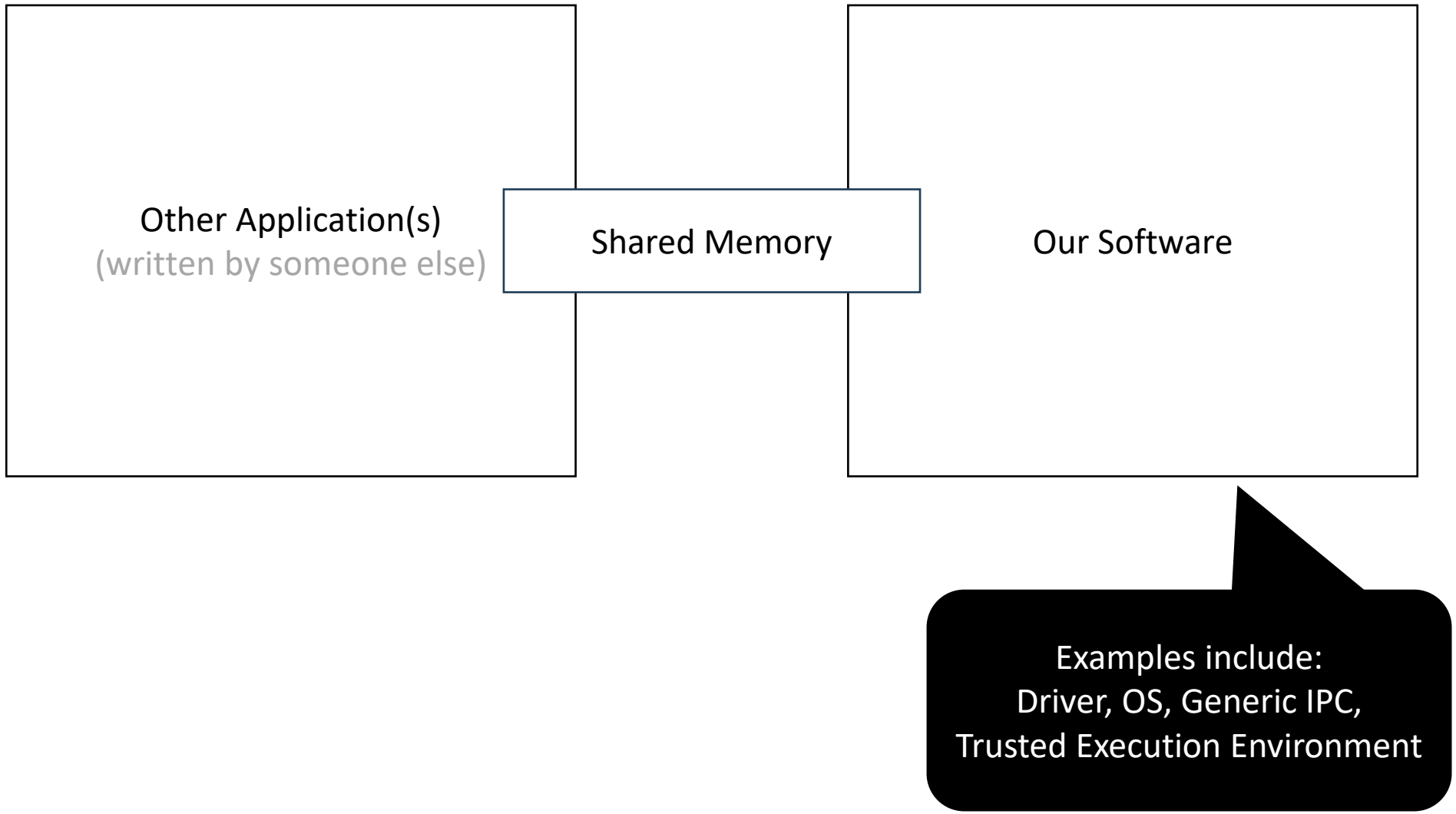
Update Statistics

Send to Billboard


C++ rocks!

"C++ sucks!"


Message Receiver Application
Update Text Command: "C++ sucks!"
Shared Memory
Friendly Billboard™ Firmware
Is Friendly?
Update Statistics
STOP


Other Application(s)
(written by someone else)
Shared Memory
Our Software
Examples include:
Driver, OS, Generic IPC,
Trusted Execution Environment

```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```

```cpp
enum class Status {
    waiting,
    accepted,
    rejected
};
```

Friendly Billboard™ Firmware

```cpp
auto received = shared_memory::deserialize<ShowMessageCommand>();
process_message_command(received.message, *received.status);
```

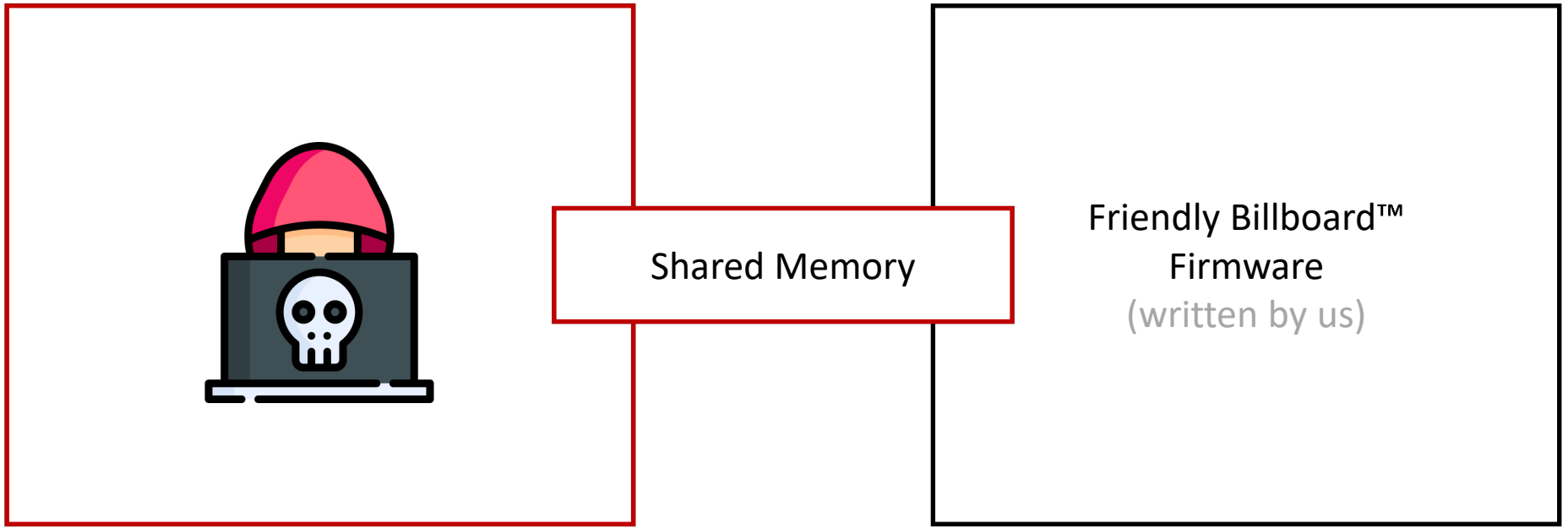
```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```

```cpp
enum class Status {
    waiting,
    accepted,
    rejected
};
```

Friendly Billboard™ Firmware

```cpp
auto received = shared_memory::deserialize<ShowMessageCommand>();
process_message_command(received.message, *received.status);
```

```cpp
void process_message_command(std::string_view message, Status& status)
{
    status = is_friendly(message) ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(message); }
}
```



Attacker Model:

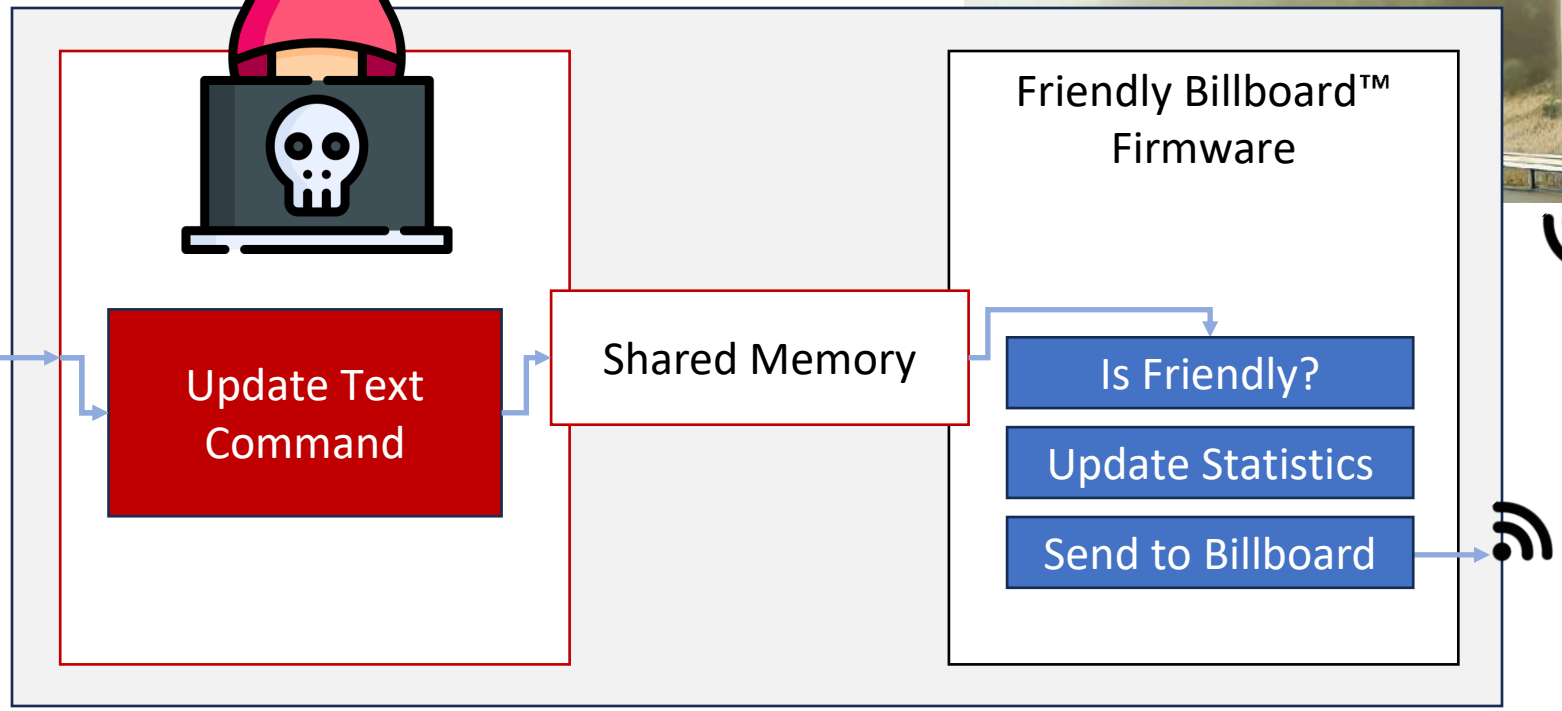
Attacker has taken over the message receiver.

Attacker Goal:

Display an unfriendly message on the billboard.


WWW


C++ sucks!


Update Text Command
Shared Memory
Friendly Billboard™ Firmware
Is Friendly?
Update Statistics
Send to Billboard

```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```

Friendly Billboard™ Firmware

```cpp
auto received = shared_memory::deserialize<ShowMessageCommand>();
process_message_command(received.message, *received.status);
```

```cpp
void process_message_command(std::string_view message, Status& status)
{
    status = is_friendly(message) ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(message); }
}
```

CWE-20	Improper Input Validation

Validate/Verify

```
1  if (input > MAX_ALLOWED_VALUE)
2  {
3      return ERROR_CODE;
4  }
```

```
1  T input = ...; // input is untrusted!!
```

Sanitize

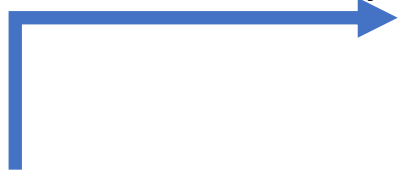
```
1  if (input > MAX_ALLOWED_VALUE)
2  {
3      input = MAX_ALLOWED_VALUE;
4  }
```

```cpp
void process_message_command(std::string_view message, Status& status)
{
    status = is_friendly(message) ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(message); }
}
```

1. Raw value always accessible → possible to forget validation

2. How do I even know that I have to validate **here**?

Validate/Verify



```
1  std::optional<T> verified = input.verify(
2      [](auto x) { return x <= MAX_ALLOWED_VALUE; }
3  );
```

```
1  Untrusted<T> input = ...;
```

Sanitize

```
1  T sanitized = input.sanitize(
2      [](auto x) { return std::min(x, MAX_ALLOWED_VALUE); }
3  );
```

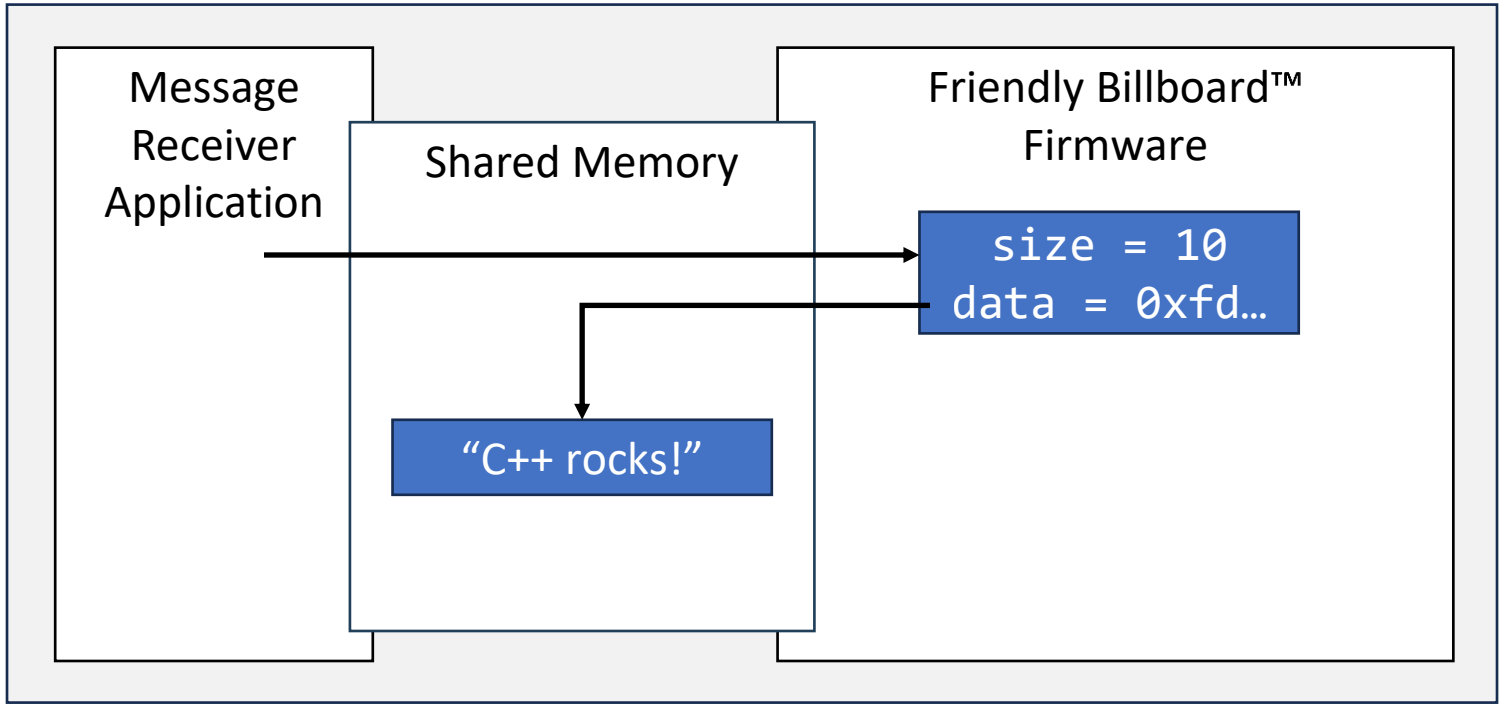
```cpp
template<typename T>
class Untrusted
{
    T m_value;

public:
    // + constructor, copy, move operations

    template<typename Func>
    auto sanitize(Func&& sanitizer)
    {
        return sanitizer(m_value);
    }

    template<typename Func>
    std::optional<T> verify(Func&& predicate)
    {
        return predicate(m_value) ? std::optional{m_value} : std::nullopt;
    }
};
```

```cpp
void process_message_command(Untrusted<std::string_view> message, Status& status)
{
    auto verified = message.verify(is_friendly);
    status        = verified.has_value() ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(*verified); }
}
```

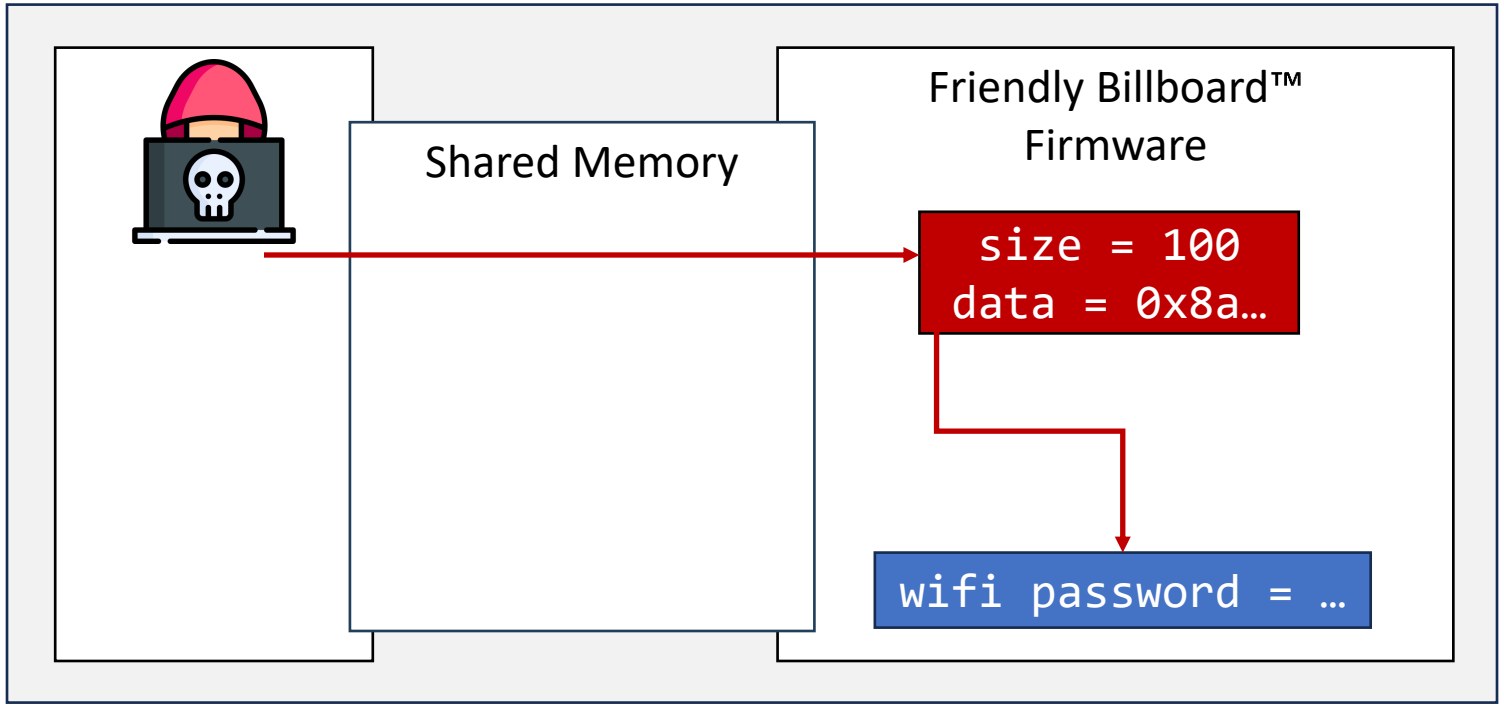
1. Raw value never accessible
2. Impossible to forget validation
3. It is clear whether some validation is still needed

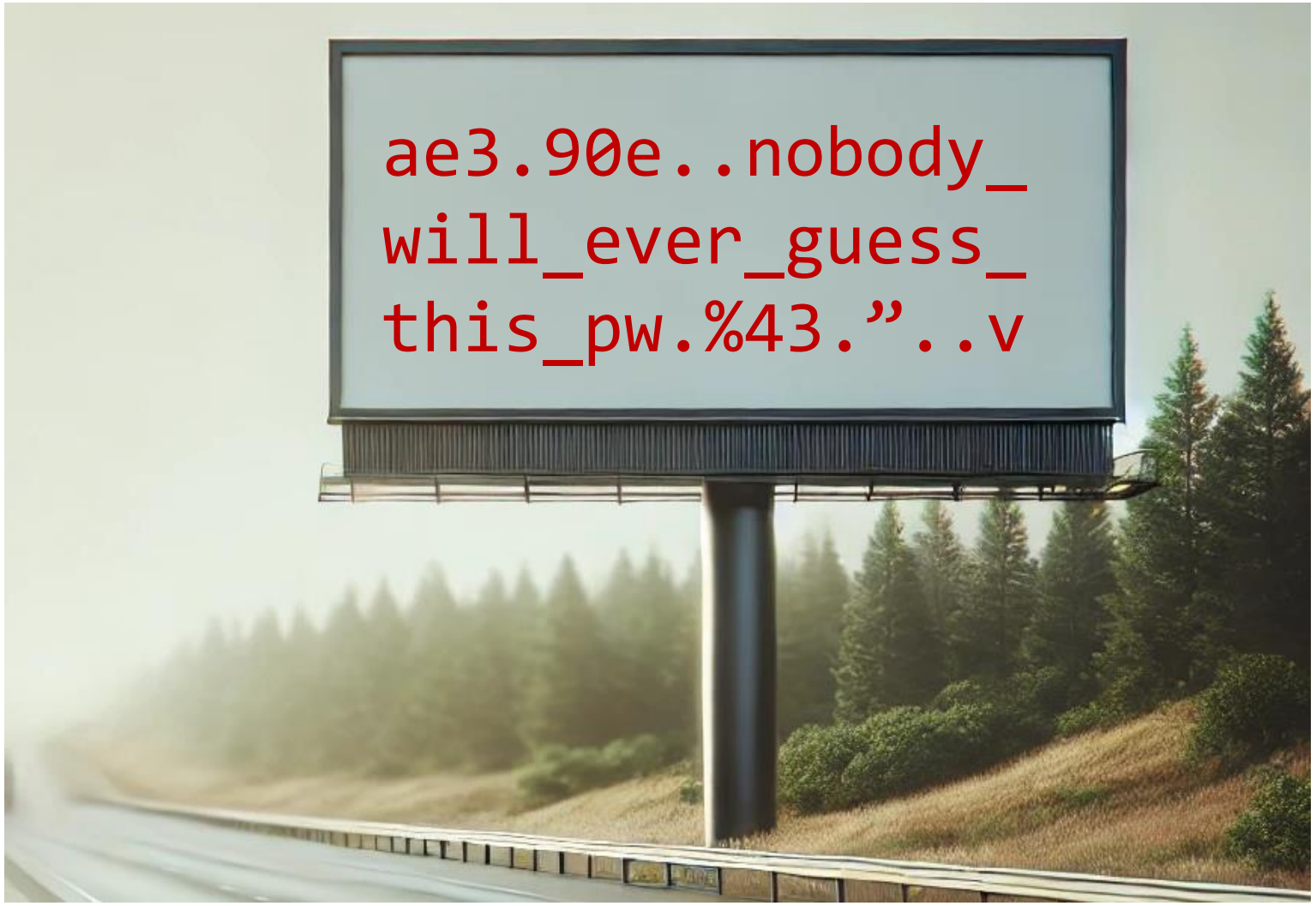
"Hm okay, but we already performed this validation, so… why the effort?"

```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```



```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```




ae3.90e..nobody_
will_ever_guess_
this_pw.%43.”..v

```cpp
struct ShowMessageCommand
{
    std::string_view message;    // [input]  text to display
    Status* status;              // [output] reports status to user
};
```

```cpp
struct ShowMessageCommand
{
    const char* message;    // [input]  text to display
    size_t length;          // [input]  length of text to display
    Status* status;         // [output] reports status to user
};
```

```cpp
// api.h
struct ShowMessageCommand
{
    const char* message;    // [input]  text to display
    size_t length;          // [input]  length of text to display
    Status* status;         // [output] reports status to user
};
```

```cpp
// internal/api.h
namespace untrusted
{
    struct ShowMessageCommand
    {
        Untrusted<const char*> message;
        Untrusted<size_t> length;
        Untrusted<Status*> status;
    };
}
```

Autogenerated

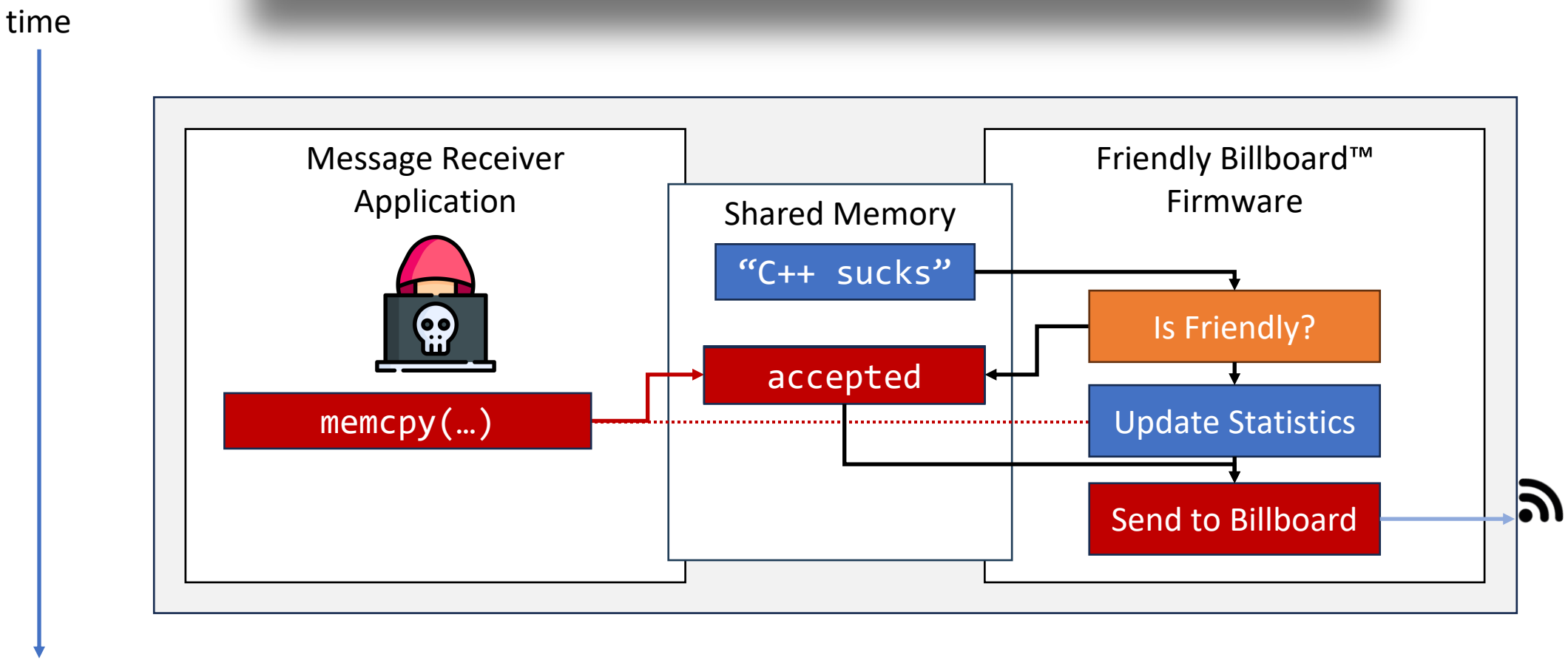
```cpp
auto received = shared_memory::deserialize<ShowMessageCommand>();
process_message_command(received.message, *received.status);
```

```cpp
auto received = shared_memory::deserialize<untrusted::ShowMessageCommand>();

auto message = received.message.verify(is_in_shared_memory);
auto length  = received.length.sanitize(truncate_to_max_text_length);
auto status  = received.status.verify(is_in_shared_memory);

if (message.has_value() && status.has_value()) {
    Untrusted<std::string_view> partially_valid(message.value(), length);
    process_message_command(partially_valid, *status.value());
} else if (status.has_value()) {
    *status.value() = Status::rejected;
}
```

```cpp
auto received = shared_memory::deserialize<untrusted::ShowMessageCommand>();

auto message = received.message.verify(is_in_shared_memory);
auto length  = received.length.sanitize(truncate_to_max_text_length);
auto status  = received.status.verify(is_in_shared_memory);

if (message.has_value() && status.has_value()) {
    Untrusted<std::string_view> partially_valid(message.value(), length);
    process_message_command(partially_valid, *status.value());
} else if (status.has_value()) {
    *status.value() = Status::rejected;
}
```

```cpp
void process_message_command(Untrusted<std::string_view> message, Status& status)
{
    auto verified = message.verify(is_friendly);
    status        = verified.has_value() ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(*verified); }
}
```

```cpp
void process_message_command(Untrusted<std::string_view> message, Status& status)
{
    auto verified = message.verify(is_friendly);
    status        = verified.has_value() ? Status::accepted : Status::rejected;
    update_statistics(status);
    if (status == Status::accepted) { show_on_billboard(*verified); }
}
```

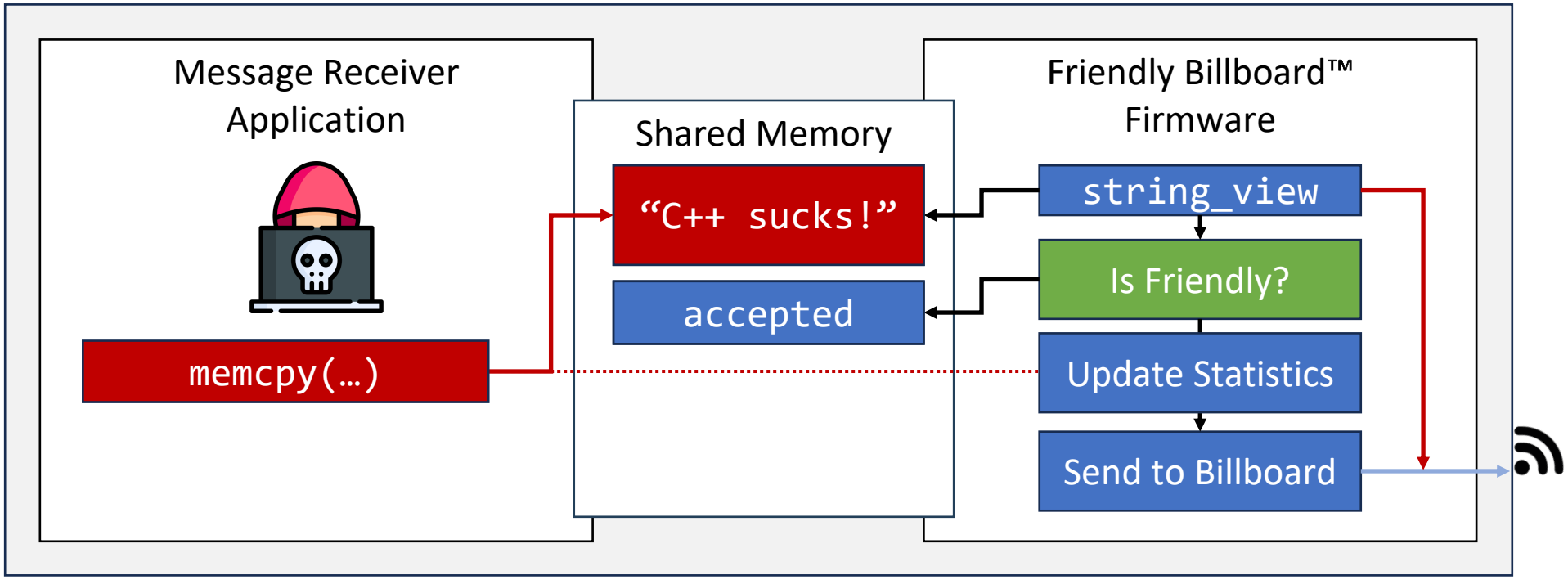


```cpp
template<typename T>
class WriteOnly
{
    T* m_pointer;

public:
    WriteOnly(T* pointer) : m_pointer{pointer} {}

    void operator=(const T& t) { *m_pointer = t; }
};
```

```cpp
void process_message_command(Untrusted<std::string_view> message, WriteOnly<Status> status)
{
    auto verified     = message.verify(is_friendly);
    auto local_status = verified.has_value() ? Status::accepted : Status::rejected;
    status            = local_status;
    update_statistics(local_status);
    if (local_status == Status::accepted) { show_on_billboard(*verified); }
}
```

```cpp
void process_message_command(Untrusted<std::string_view> message, WriteOnly<Status> status)
{
    auto verified     = message.verify(is_friendly);
    auto local_status = verified.has_value() ? Status::accepted : Status::rejected;
    status            = local_status;
    update_statistics(local_status);
    if (local_status == Status::accepted) { show_on_billboard(*verified); }
}
```

time



Time of Check vs Time of Use (TOCTOU)

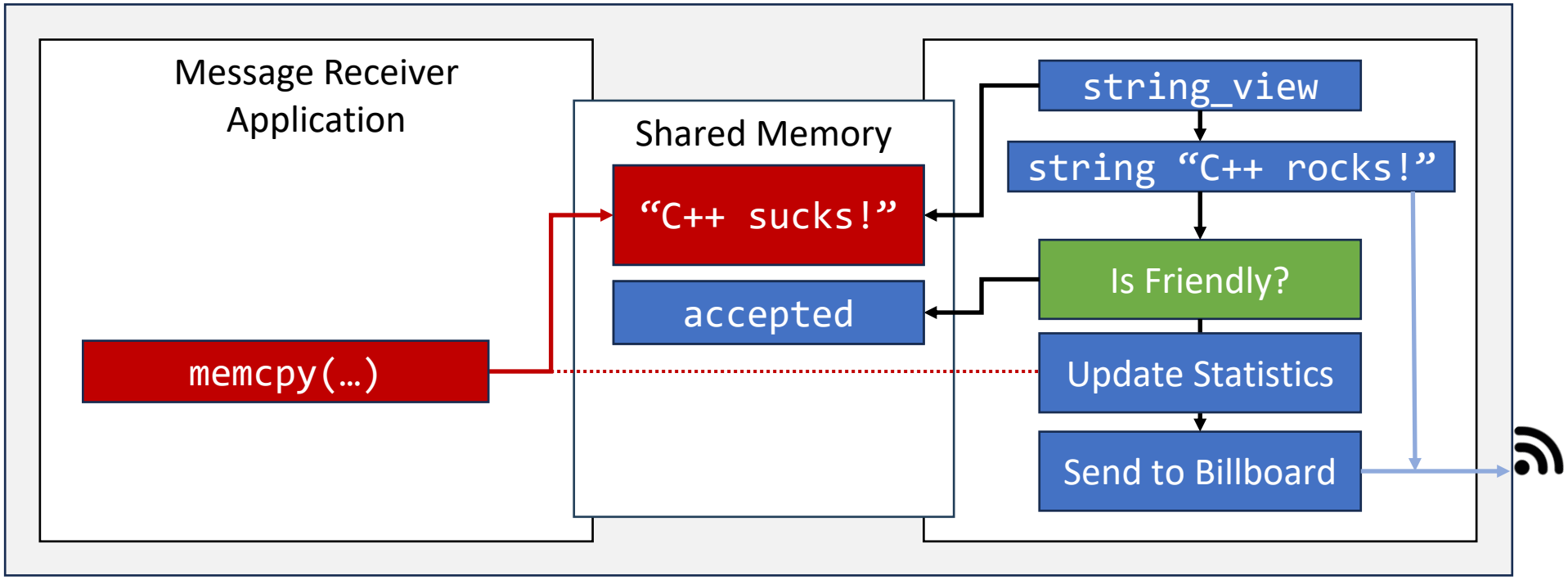
Vulnerability

```cpp
auto received = shared_memory::deserialize<untrusted::ShowMessageCommand>();

auto message = received.message.verify(is_in_shared_memory);
auto length  = received.length.sanitize(truncate_to_max_text_length);
auto status  = received.status.verify(is_in_shared_memory);

if (message.has_value() && status.has_value()) {
    Untrusted<std::string_view> partially_valid(message.value(), length);
    WriteOnly wrapped_status{status.value()};
    process_message_command(partially_valid, wrapped_status);
} else if (status.has_value()) {
    *status.value() = Status::rejected;
}
```

```cpp
auto received = shared_memory::deserialize<untrusted::ShowMessageCommand>();

auto message = received.message.verify(is_in_shared_memory);
auto length  = received.length.sanitize(truncate_to_max_text_length);
auto status  = received.status.verify(is_in_shared_memory);

if (message.has_value() && status.has_value()) {
    Untrusted<std::string> partially_valid(message.value(), length);
    WriteOnly wrapped_status{status.value()};
    process_message_command(partially_valid, wrapped_status);
} else if (status.has_value()) {
    *status.value() = Status::rejected;
}
```

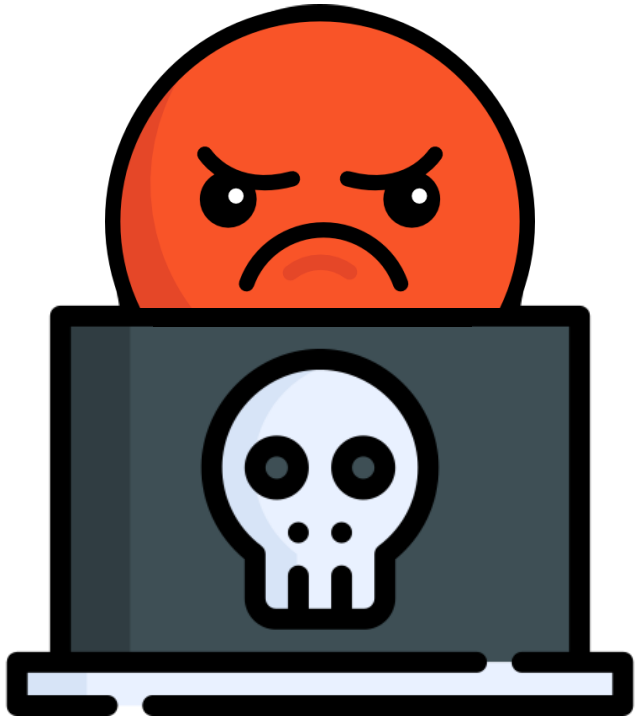
```cpp
void process_message_command(Untrusted<std::string>& message, WriteOnly<Status> status)
{
    auto verified     = message.verify(is_friendly);
    auto local_status = verified.has_value() ? Status::accepted : Status::rejected;
    status            = local_status;
    update_statistics(local_status);
    if (local_status == Status::accepted) { show_on_billboard(*verified); }
}
```

time



```cpp
auto received = shared_memory::deserialize<untrusted::ShowMessageCommand>();

auto message = received.message.verify(is_in_shared_memory);
auto length  = received.length.sanitize(truncate_to_max_text_length);
auto status  = received.status.verify(is_in_shared_memory);

if (message.has_value() && status.has_value()) {
    Untrusted<std::string> partially_valid(message.value(), length);
    WriteOnly wrapped_status{status.value()};
    process_message_command(partially_valid, wrapped_status);
} else if (status.has_value()) {
    *status.value() = Status::rejected;
}
```

```cpp
void process_message_command(Untrusted<std::string>& message, WriteOnly<Status> status)
{
    auto verified     = message.verify(is_friendly);
    auto local_status = verified.has_value() ? Status::accepted : Status::rejected;
    status            = local_status;
    update_statistics(local_status);
    if (local_status == Status::accepted) { show_on_billboard(*verified); }
}
```



Security is not about
what is intended,
but about
what is possible.

Arguing about security is mostly meaningless without a proper **attacker model**.

Make the **right thing easy** and the **wrong thing hard** to do.

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design

GitHub

https://github.com/not-a-trojan/secure_coding

LinkedIn

www.linkedin.com/in/max-hoffmann-6a3484254

Icons via

Freepik

Parzival’ 1997

https://www.flaticon.com/

Octopocto

Monkik

LAFS

Security Beyond Memory Safety

Using Modern C++ to Avoid Vulnerabilities by Design

MAX HOFFMANN