Rust vs. C++: Preventing Common Software Vulnerabilities

Introduction

This study investigates common classes of software vulnerabilities in C++ and compares how the Rust programming language addresses or prevents them. By analyzing specific examples in both C++ and Rust, we aim to understand how Rust's design principles—particularly its ownership system, borrow checker, and strict type safety—lead to safer code by default.

Software security vulnerabilities continue to be a significant concern in systems programming, especially in memory-unsafe languages like C and C++. Many high-profile security exploits, including buffer overflows and use-after-free bugs, stem from improper memory management, lack of bounds checking, or undefined behavior—issues which are prevalent in low-level code written without sufficient safeguards.

In recent years, the Rust programming language has gained attention for its promise of memory safety without a garbage collector, offering an alternative to C++ that enforces strict compile-time rules to eliminate entire classes of vulnerabilities. Rust's ownership system, borrow checker, and safety guarantees allow developers to write performant code with significantly reduced risk of common bugs.

Study Objective

The primary goal of this independent study is to explore how Rust addresses a set of widely known vulnerability classes that are historically difficult to avoid in C++. The study involves:

- Identifying and analyzing several common types of memory and concurrency vulnerabilities in C++
- Implementing minimal C++ examples that demonstrate each vulnerability
- Rewriting the same logic in Rust and analyzing how the language either prevents or mitigates the vulnerability
- Comparing the two implementations in terms of behavior, safety, and developer burden

This comparative analysis will help highlight how Rust enforces safer programming practices through language design and compile-time enforcement.

Vulnerability Scope

The report focuses on the following nine vulnerability classes, all of which are well-documented in real-world software failures and security advisories:

1. **Buffer Overflow**
2. **Use-After-Free**
3. **Null Pointer Dereference**

4. **Out-of-Bounds Read/Write**
5. **Double Free**
6. **Memory Leaks**
7. **Race Conditions (Data Races)**
8. **Uninitialized Memory Access**
9. **Dangling Pointers**
10. **Integer Overflow / Underflow**

These were selected based on their relevance in the CWE Top 25 Most Dangerous Software Weaknesses and their prevalence in C/C++ systems.

Methodology

Each vulnerability is demonstrated using a short C++ code snippet that triggers or exposes the issue, either through unsafe memory access or undefined behavior. The corresponding Rust example is written using safe Rust wherever possible, and is analyzed to show how the language design prevents the same mistake.

Screenshots, panic messages, compiler warnings, and runtime behavior are recorded and compared. When applicable, references to The Rust Programming Language ("The Book") and Rust documentation are included to explain specific mechanisms (e.g., borrow checker, lifetime rules, bounds checks).

All code was compiled and tested in a reproducible environment, detailed in the section titled Development and Compilation Environment.

Expected Outcome

The final deliverable is a detailed technical report that can serve as a practical reference for students and developers interested in understanding how Rust compares to C++ in the context of software security. The report includes:

- Annotated source code for both C++ and Rust
- Explanations of the vulnerabilities and how they arise
- Analysis of how Rust mitigates or prevents them
- Observations about developer experience and safety guarantees

Development and Compilation Environment

To ensure reproducibility, the following setup was used for writing, compiling, and running the C++ and Rust examples:

System Information

- Operating System: macOS Ventura (Apple Silicon / Intel)
- Shell: Terminal (zsh)
- Editor: Visual Studio Code (VS Code)

Rust Environment

- Rust Version: rustc 1.70.0 or later
- Install Source: https://rustup.rs
- Run Toolchain: cargo and rustc used via terminal
- Rust IDE Extension: rust-analyzer (official VS Code extension by rust-lang)

Compile & Run (Basic):
rustc file.rs
./file

Alternative with Cargo (Recommended for Projects):
cargo new project_name
cd project_name
cargo run

C++ Environment
- Compiler: Apple Clang (Xcode Command Line Tools)
- C++ Version: C++17
- Command Used:

g++ -std=c++17 file.cpp -o program
./program

Note: C++ warnings were enabled by default. Compiler diagnostics were helpful in pointing out some issues like out-of-bounds access, but they did not prevent compilation.

IDE Behavior (VS Code)
- Running .cpp or .rs files inside VS Code may prompt installation of debugger extensions. For this study:
- Most Rust code was run via terminal using rustc or cargo.
- C++ was compiled and executed from terminal using g++.

Additional Tools (Optional)
- lldb or gdb for debugging
- valgrind for C++ memory analysis (not available by default on macOS)
- cargo clippy and cargo check for Rust linting and static checks

This environment ensures that all code behavior (especially crashes, panics, and memory violations) are observed in a clean and verifiable way.

Reproducibility Instructions:

All C++ and Rust code examples in this report are provided in an accompanying folder in GitHub. The folder contains three sets of C++ programs demonstrating unsafe memory access patterns and their equivalent Rust programs showing safe handling or compile-time prevention.

To reproduce the results:

1. Install **g++ (C++17 or later)** and **Rust (rustc)** on your system.
2. Unzip the provided code folder.

3. Follow the compile and run commands in the included `README.txt` file.
4. For Rust Example 3, compilation will fail intentionally, demonstrating Rust's compile-time bounds checking.
5. All examples were tested on macOS 15, but should also work on Linux with no modifications. On Windows, use WSL or a similar Unix-like environment for running the shell commands.

Vulnerability Cases

1. Buffer Overflow

Description

A buffer overflow occurs when a program writes more data to a buffer (a fixed-size block of memory) than it can hold. In C and C++, where array bounds are not automatically checked at runtime, this can lead to data corruption, unexpected behavior, or security vulnerabilities such as arbitrary code execution.

In contrast, Rust performs bounds checking and panics safely at runtime in safe code, preventing such memory violations. Rust can also prevent some out-of-bounds accesses at **compile time** if the index is known to be constant.

Example 1: Out-of-Bounds Integer Array Write

C++ Version – Unsafe Memory Write

```cpp
// buffer_overflow_1.cpp
#include <iostream>

int main() {
    int buffer[5] = {0};           // Line 4: fixed-size buffer (valid indices 0..4)
    buffer[10] = 42;               // Line 5: DANGEROUS -- out-of-bounds write (undefined behavior)
    std::cout << "Buffer: " << buffer[10] << std::endl; // Line 6: reads the out-of-bounds slot
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The C++ language does **not** mandate runtime bounds checks for built-in arrays, and most compilers (Clang/GCC/MSVC) **do not insert** checks by default. The standard defines out-of-bounds access as **undefined behavior (UB)**, so the compiler is allowed to assume it never happens and generate code accordingly.

- **What is dangerous:**
    - **Line 5** writes to `buffer[10]`, which is beyond the allocated object (`buffer[0..4]`). This can overwrite adjacent data on the stack.
    - **Line 6** then **reads** from the same invalid element, which may print a "plausible" value, print garbage, or crash—**the behavior is not defined** and can vary by compiler flags, optimization level, stack layout, and OS.

- **Real-world consequence:** UB can enable exploits (e.g., smashing return addresses or control data), cause intermittent crashes, or silently corrupt program state.

Rust Version – Safe Array Access with Panic

```
// buffer_overflow_1.rs
fn main() {
    let buffer = [0; 5];                // Line 3: fixed-size array (valid
indices 0..4)
    let value = buffer[10];             // Line 4: DANGEROUS in logic --
triggers a bounds check and panics at runtime
    println!("Buffer: {}", value);
}
```

Explanation (Rust):

- **How it is prevented:** Rust **always** inserts a bounds check on safe indexing.
    - **Line 4** triggers a **panic** with a clear message (e.g., "index out of bounds: the len is 5 but the index is 10").
- **Why this is safer than C++:** Instead of continuing with memory corruption, Rust **stops the program predictably** at the point of violation. You can also avoid panics by using `.get()` (Example 3).

Full Terminal Example – Buffer Overflow (Example 1)

The following real terminal sessions demonstrate how C++ and Rust handle the same buffer overflow scenario.
C++: Compilation Warning, Undefined Behavior at Runtime

```
(base) xs-air:buffer_overflow_section_pkg xhmac$ g++ -std=c++17 buffer_overflow_1.cpp -o output
buffer_overflow_1.cpp:4:5: warning: array index 10 is past the end of the array (that has type 'int[5]') [-Warray-bounds]
    4 |     buffer[10] = 42;              // out-of-bounds write
      |     ^      ~~
buffer_overflow_1.cpp:3:5: note: array 'buffer' declared here
    3 |     int buffer[5] = {0};          // fixed-size buffer
      |     ^
buffer_overflow_1.cpp:5:32: warning: array index 10 is past the end of the array (that has type 'int[5]') [-Warray-bounds]
    5 |     std::cout << "Buffer: " << buffer[10] << std::endl;
      |                                ^      ~~
buffer_overflow_1.cpp:3:5: note: array 'buffer' declared here
    3 |     int buffer[5] = {0};          // fixed-size buffer
      |     ^
2 warnings generated.
(base) xs-air:buffer_overflow_section_pkg xhmac$ ./output
Buffer: 42
Bus error: 10
```

Compilation Output:

- Compiler issues **warnings** about accessing an array index past the end.
- No compilation failure—execution is still allowed.

The program writes past the buffer's boundary, corrupting memory and crashing with a bus error. This is **undefined behavior**—results could vary depending on compiler, system, and memory layout.

Rust Case 1 – Compile-Time Detection

```
(base) xs-air:buffer_overflow_section_pkg xhmac$ rustc buffer_overflow_1.rs
error: this operation will panic at runtime
 --> buffer_overflow_1.rs:3:17
  |
3 |     let value = buffer[10];              // causes panic at runtime
  |                 ^^^^^^^^^^ index out of bounds: the length is 5 but the index is 10
  |
  = note: `#[deny(unconditional_panic)]` on by default

error: aborting due to 1 previous error

(base) xs-air:buffer_overflow_section_pkg xhmac$ ./buffer_overflow_1.rs
-bash: ./buffer_overflow_1.rs: Permission denied
```

If Rust can determine at compile time that an index is invalid (e.g., `buffer[10]` on a fixed `[0; 5]` array), compilation fails with:

error: this operation will panic at runtime
index out of bounds: the length is 5 but the index is 10

Rust Case 2 – Runtime Panic with Backtrace

When the invalid index is determined **at runtime** instead of compile time, Rust compiles the program but halts execution predictably with a panic.

```
(base) xs-air:buffer_overflow_section_pkg xhmac$ rustc buffer_overflow_1runtime.rs
(base) xs-air:buffer_overflow_section_pkg xhmac$ ./buffer_overflow_1runtime

thread 'main' panicked at buffer_overflow_1runtime.rs:11:17:
index out of bounds: the len is 5 but the index is 10
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
(base) xs-air:buffer_overflow_section_pkg xhmac$ RUST_BACKTRACE=1 ./buffer_overflow_1runtime 10

thread 'main' panicked at buffer_overflow_1runtime.rs:11:17:
index out of bounds: the len is 5 but the index is 10
stack backtrace:
   0: __rustc::rust_begin_unwind
             at /rustc/6b00bc3880198600130e1cf62b8f8a93494488cc/library/std/src/panicking.rs:697:5
   1: core::panicking::panic_fmt
             at /rustc/6b00bc3880198600130e1cf62b8f8a93494488cc/library/core/src/panicking.rs:75:14
   2: core::panicking::panic_bounds_check
             at /rustc/6b00bc3880198600130e1cf62b8f8a93494488cc/library/core/src/panicking.rs:281:5
   3: buffer_overflow_1runtime::main
   4: core::ops::function::FnOnce::call_once
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

Running with a backtrace: `RUST_BACKTRACE=1 ./buffer_overflow_1runtime 10` produces a detailed stack trace pinpointing the error location.

Example 2: Character Buffer Overflow

C++ Version – Writing Beyond Buffer Size

```cpp
// buffer_overflow_2.cpp
#include <iostream>
using namespace std;

int main() {
    char buffer[5];                     // Line 6: capacity is 5 bytes (0..4)
    for (int i = 0; i < 10; ++i) {      // Line 7: loop exceeds the valid range
        buffer[i] = 'A';                // Line 8: DANGEROUS -- writes past index 4 (UB for i >= 5)
    }
    buffer[4] = '\0';                   // Line 10: tries to null-terminate last valid slot
    cout << "Buffer content: " << buffer << endl;
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** As with Example 1, **no mandatory bounds checks** exist for C-style arrays. The compiler does not stop the program because the standard treats this as UB, and compilers typically prioritize performance over checks.

- **What is dangerous:**
    - **Line 8** performs repeated **out-of-bounds writes** for `i = 5..9`, corrupting adjacent memory.
    - Resulting behavior can include truncated prints, strange characters, or a crash—again **undefined**.

Rust Version – Safe Character Array Write

```
// buffer_overflow_2.rs
fn main() {
    let mut buffer = [0u8; 5];          // Line 3: capacity is 5 bytes (0..4)
    for i in 0..10 {                    // Line 4: loop exceeds the valid range
        buffer[i] = b'A';               // Line 5: DANGEROUS in logic -- panics at i == 5 (bounds check)
    }
    println!("Buffer: {:?}", buffer);
}
```

Explanation (Rust):

- **What happens:** On **Line 5**, as soon as `i == 5`, indexing fails the bounds check and **panics**.
- **Why it's safe:** The program never writes beyond allocated memory; it **halts** at the attempted violation with a precise error.

Example 3: Safe Access Using `.get()` in Rust (and Compile-Time Prevention)

Rust Version – Compile-Time Error (Constant Index)

```
// buffer_overflow_3_compile.rs
fn main() {
    let buffer = [1, 2, 3, 4, 5];
    let value = buffer[10]; // Line 4: DANGEROUS -- known-constant OOB; compiler errors (when proven at compile time)
```

```
    println!("Value: {}", value);
}
```

Rust Compile Output (illustrative):

```
error: index out of bounds: the length is 5 but the index is 10
 --> buffer_overflow_3_compile.rs:4:17
  |
4 |     let value = buffer[10];
  |                 ^^^^^^^^^^
```

Rust Version – Optional Index Access

```
// buffer_overflow_3.rs
fn main() {
    let buffer = [1, 2, 3, 4, 5];
    match buffer.get(10) {             // Line 4: SAFE -- returns
Option<&T>
        Some(val) => println!("Value: {}", val),
        None => println!("Index out of bounds"),
    }
}
```

Explanation (Rust):

- **Compile-time protection:** When the compiler can **prove** the index is out of bounds at compile time (e.g., a known constant index), it **rejects** the code.
- **Runtime safety with `.get()`:** Using `.get()` returns an `Option`, forcing explicit handling of the out-of-bounds case **without panic**.

No C++ Equivalent:
C++ provides **no built-in** checked indexing for raw arrays; you must either write manual checks or use safer containers (e.g., `std::vector::at()`), which **throws** on OOB but is not used with raw arrays.

CWE Mapping

- **CWE-787:** [Out-of-Bounds Write]

- **CWE-120:** [Buffer Copy without Checking Size]()
- **CWE-119:** [Improper Restriction of Operations within the Bounds of a Memory Buffer]()

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Array bounds checking	None; UB at **Ex1:L5**, **Ex2:L8**	Checked; panic at **Ex1:L4**, **Ex2:L5**
Compile-time prevention	Not applicable (raw arrays)	Compile-time error at **Ex3:L4** (constant OOB)
Runtime behavior	May corrupt/crash or “seem fine”	Deterministic panic or safe `Option` path
Developer surface	Manual discipline/tools required	Safe defaults; `.get()` for non-panicking access

---

Developer Takeaway

In C++, out-of-bounds writes are a frequent source of memory corruption and critical security vulnerabilities. Because the language lacks built-in array bounds checking, developers must manually prevent these bugs.
Rust prevents such vulnerabilities through strict bounds checking in safe code. Attempts to read or write out of bounds will either:

- **Fail to compile** (when the index is a constant the compiler can analyze),
- **Panic at runtime** with an informative error, or
- **Return an `Option`** (when using `.get()`).
  This ensures a safer and more predictable development experience and drastically reduces common buffer-handling bugs.

Security Implications

Buffer overflows are one of the most critical and historically exploited memory vulnerabilities, often leading to program crashes, silent data corruption, and severe security exploits such as arbitrary code execution or privilege escalation. In C and C++, the absence of built-in runtime bounds checking allows writes beyond the buffer’s limits to go undetected at compile time and frequently even at runtime, making it possible for attackers to overwrite control structures, manipulate program flow, or inject malicious payloads.

Rust eliminates this vulnerability in safe code through strict array bounds checking enforced at runtime, and in many cases, at compile time when the index is a known constant. Any attempt to access memory outside the allocated range triggers a controlled panic, halting execution predictably rather than risking undefined behavior. Combined with Rust's ownership and borrowing rules, this ensures memory safety without relying solely on developer discipline or external tooling, effectively removing buffer overflows as a class of bugs in safe Rust code.

Reproducibility Note
All example files from this section are available in the downloadable folder:
buffer_overflow_section_pkg.zip.
To replicate: (a) install **g++ (C++17+)** and **Rust (rustc)**, (b) unzip the folder, and (c) follow `README.txt` for build/run commands. Rust Example 3 has a **compile-time error by design** to demonstrate static protection.

---

2. Use-After-Free (UAF)

Description
Use-after-free occurs when a program continues to access memory **after it has been deallocated**. In C/C++, this is undefined behavior (UB): the program may crash, appear to work, or be exploitable. In **safe Rust**, the ownership system and borrow checker make use-after-free impossible to express: the compiler rejects code that might access freed memory.

Note on compilation/runtime environment: Build and run commands, tool versions, and platform details are provided once in the **Development and Compilation Environment** section.

Example 1: Delete Then Use

C++ Version – Free then Dereference

```
// uaf_example1_delete_then_use.cpp
#include <iostream>
int main() {
    int* p = new int(42);      // Line 3: allocate on heap
    delete p;                   // Line 4: free
    std::cout << *p << "\n";   // Line 5: DANGEROUS -- use-after-free
(undefined behavior)
    return 0;
```

```
}
```

Explanation (C++):

- **Why it compiles:** C++ does not track pointer validity after `delete`; dereferencing a freed pointer is **UB**, and the compiler is allowed to assume it never happens.

- **What is dangerous:**
    - **Line 4** frees the storage.
    - **Line 5** dereferences `p` **after free**. The program may print a stale value, crash, or corrupt memory, depending on allocator and optimization.

Rust Version – Use After Move is Rejected

```
// uaf_example1_move_then_use.rs
fn main() {
    let b = Box::new(42);   // Line 2: allocate
    let moved = b;          // Line 3: move ownership to `moved`; `b` is
now invalid
    // println!("{}", b);   // Line 4: COMPILE ERROR -- use of moved value
`b`
    println!("{}", moved);  // Line 5: OK: value used by its new owner
}
```

Explanation (Rust):

- **How it's prevented:** Ownership moves make the previous binding (**b**) **unusable**. If you uncomment **Line 4**, the compiler errors ("use of moved value"), preventing any use-after-free pattern in safe code.
- **Why this matters:** Rust encodes lifetime/ownership in the type system, so the compiler stops UAF before the program runs.

Example 2: Returning Address of a Local (Dangling)

C++ Version – Dangling Pointer via Function Return

```
// uaf_example2_return_local_address.cpp
#include <iostream>
int* make_ptr() {
```

```
    int x = 99;                     // Line 3: stack local
    return &x;                      // Line 4: DANGEROUS -- returns address to dead stack object
}
int main() {
    int* p = make_ptr();
    std::cout << *p << "\n";    // Line 8: use-after-lifetime / dangling pointer (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The C++ type system doesn't encode lifetimes for raw pointers; the compiler can't prove the pointer escapes the variable's lifetime.
- **What is dangerous:**
  - **Line 4** returns a pointer to a stack variable that is destroyed when `make_ptr` returns.
  - **Line 8** dereferences a **dangling** pointer → **UB**.

Rust Version – Borrow Checker Rejects It

```
// uaf_example2_borrow_checker.rs
fn main() {
    let r;
    {
        let x = 99;             // Line 5: `x` lives only in this inner scope
        r = &x;                 // Line 6: borrow of `x`
    }                            // Line 7: `x` dropped here
    // println!("{}", r);        // Line 8: COMPILE ERROR -- `x` does not live long enough
}
```

Explanation (Rust):

- **How it's prevented:** The borrow checker tracks lifetimes; the reference `r` would outlive `x`. If you uncomment **Line 8**, the compiler emits an error like "borrowed value does not live long enough."
- **Why it's safer:** Rust refuses to compile code that would create **dangling references**.

Example 3: Double Free (and UAF) vs. Single Drop

C++ Version – Double Free

```cpp
// uaf_example3_double_free.cpp
#include <cstdlib>
int main() {
    int* p = (int*)std::malloc(sizeof(int)); // Line 3: allocate
    std::free(p);                             // Line 4: free
    std::free(p);                             // Line 5: DANGEROUS -- double free (UB, often exploitable)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The compiler can't track that `p` was already freed; calling `free` twice is **UB**.
- **Security impact:** Double free is a common primitive for **heap exploitation** (use-after-free, tcache poisoning, etc.).

Rust Version – One Owner, One Drop

```rust
// uaf_example3_single_drop.rs
fn main() {
    let p = Box::new(7);
    drop(p);                    // Line 3: value is moved into drop; memory freed exactly once
    // drop(p);                 // Line 4: COMPILE ERROR -- value used after move (prevents double free)
}
```

Explanation (Rust):

- **How it's prevented:** Ownership ensures exactly one drop. Attempting to drop twice (uncomment **Line 4**) is a compile error ("use of moved value").
- **Why it's safe:** No double free, therefore no UAF via stale pointer.

(Optional) Example 4: Unsafe Raw Pointers in Rust

```rust
// uaf_example4_unsafe_raw_ptr.rs
// Demonstration only -- do not run.
fn main() {
    let b = Box::new(5);
    let raw = Box::into_raw(b);      // take ownership as raw pointer
    unsafe {
        Box::from_raw(raw);          // free once
        // println!("{}", *raw);     // would be UAF if uncommented
(undefined behavior)
    }
}
```

Explanation:

- Safe Rust prevents UAF. Only **`unsafe`** raw pointer manipulation can reintroduce it, which is explicitly opt-in and audited.

CWE Mapping

- **CWE-416:** [Use After Free]
- **CWE-415:** [Double Free]
- **CWE-562:** [Return of Stack Variable Address]

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Use after free	Allowed; UB at **Ex1:L5**	Prevented by move semantics; compile error at **Ex1:L4**
Dangling pointer / lifetime	Allowed; UB at **Ex2:L8**	Prevented by borrow checker; compile error at **Ex2:L8**
Double free	Allowed; UB at **Ex3:L5**	Prevented; compile error at **Ex3:L4**
Boundary of safety	Not enforced by language	Safe Rust enforces; UAF only possible with explicit `unsafe`

Developer Takeaway
In C/C++, **use-after-free** stems from manual memory management with no language-level lifetime tracking; the compiler accepts code that can later **dereference freed or dangling memory**. In **safe Rust**, the type system encodes ownership and lifetimes, so UAF patterns either **don't type-check** (compile-time error) or **cannot be expressed** without `unsafe`. This removes an entire class of bugs before the code runs.

Security Implications
UAF is a high-severity vulnerability class widely exploited in real systems (privilege escalation, arbitrary code execution). Rust's ownership and borrow checking eliminate UAF in safe code by construction. The only way to reintroduce UAF is via explicit `unsafe` and raw pointer manipulation, which isolates risk and encourages audits.

Reproducibility Note
All example files from this section are available in the downloadable folder:
use_after_free_section_pkg.zip — Download

To replicate:
1. Install **g++ (C++17 or later)** and **Rust (rustc)**.
2. Unzip the folder; see `README.txt` for compile/run commands.
3. Uncomment the noted lines in the Rust examples to see the **compile-time errors** the report references.

---

3. Null Pointer Dereference

Description
A null pointer dereference happens when a program attempts to access memory through a pointer whose value is `NULL`/`nullptr`. In C/C++, this results in a segmentation fault or other undefined behavior. **Safe Rust** avoids null references entirely by using `Option<T>` to model "maybe a value," and by requiring references to be initialized and valid at compile time.

Note:

C++ null pointer dereference vulnerabilities generally arise from the same fundamental unsafe access pattern, often differing only in context (e.g., dereferencing after failed allocation, dereferencing uninitialized pointers). To avoid redundancy, only representative C++ cases are included here, while multiple Rust examples are shown to illustrate the various safety mechanisms.

Example 1: Classic Null Dereference

C++ Version – Dereferencing `nullptr`

```cpp
// npd_example1_null_deref.cpp
#include <iostream>
int main() {
    int* p = nullptr;              // Line 3: null pointer
    std::cout << *p << "\n";       // Line 4: DANGEROUS -- null dereference (segfault/UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The language allows `nullptr` for raw pointers and does not require dereference checks. Dereferencing `p` is **undefined behavior (UB)** and typically crashes with a segmentation fault at runtime.

- **What is dangerous:**
    - **Line 4** dereferences a null pointer; behavior is not defined by the standard, so the process is typically terminated by the OS.

Rust Version – Model Absence with `Option`

```rust
// npd_example1_option_none.rs
fn main() {
    let ptr: Option<&i32> = None;      // Line 2: explicit absence (no null references)
    match ptr {                        // Line 3: must handle both cases
        Some(v) => println!("{}", v),
        None => println!("Pointer is null (None)"),
    }
}
```

Explanation (Rust):

- **How it's prevented:** Rust references are never null in safe code. “No value” is represented as `Option<T>`. Pattern matching enforces explicit handling of the `None` case, so there is no UB or crash.

Example 2: Uninitialized vs. Safe Initialization

C++ Version – Uninitialized Pointer Dereference

```cpp
// npd_example2_uninitialized_ptr.cpp
#include <iostream>
int main() {
    int* p;                         // Line 3: uninitialized (indeterminate)
    std::cout << *p << "\n";        // Line 4: DANGEROUS -- UB: dereferencing indeterminate pointer
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The C++ type system doesn't require pointers to be initialized before use. Dereferencing an indeterminate pointer is **UB**; the program may crash or print "garbage."

Rust Version – References Must Be Initialized

```rust
// npd_example3_no_null_refs.rs
fn main() {
    let r: &i32;                    // Line 2: declared, but uninitialized
    // println!("{}", r);           // Line 3: COMPILE ERROR -- use of possibly-uninitialized `r`
}
```

Explanation (Rust):

- **How it's prevented:** Rust **forbids** using an uninitialized reference. Uncommenting the `println!` produces a **compile-time error**, preventing UB before execution.

Example 3: Panic When Forcing a Value (Contrast Case)

Rust Version – Unwrapping `None` Panics

```rust
// npd_example2_unwrap_panic.rs
fn main() {
    let ptr: Option<i32> = None;       // Line 2: no value present
    println!("{}", ptr.unwrap());       // Line 3: RUNTIME PANIC -- unwrap on None
```

```
}
```

Explanation (Rust):

- **What happens:** Calling `unwrap()` on `None` **panics** with a clear message. This is **controlled failure**, not UB; the runtime stops safely rather than dereferencing null memory.

(Optional) Example 4: Raw Pointers Require `unsafe`

```
// npd_example4_unsafe_raw_ptr.rs
// Demonstration only -- do not run.
fn main() {
    let p: *const i32 = std::ptr::null(); // null raw pointer
    unsafe {
        // println!("{}", *p);           // DANGEROUS -- null deref via raw pointer (UB)
    }
}
```

Explanation:

- Safe Rust prevents null deref entirely. Only explicit `unsafe` with raw pointers can approximate C/C++ behavior; such code is outside Rust's safety guarantees and should be audited.

CWE Mapping

- **CWE-476:** NULL Pointer Dereference
- **CWE-457:** Use of Uninitialized Variable
- **CWE-824**: Access of Uninitialized Pointer
- **CWE-825**: Expired Pointer Dereference

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Null pointer dereference	Allowed; UB at **Ex1:L4**	Modeled with `Option`; safe match at **Ex1:L3–6**
Uninitialized reference/pointer	Allowed; UB at **Ex2:L4**	Compile-time error at **Ex2 (Rust):L3**
Forcing absent value	N/A	Controlled panic when `unwrap()` on `None`
Boundary of safety	Not enforced by language	Safe Rust enforces; raw null deref only via `unsafe`

Developer Takeaway
C/C++ permit dereferencing null or uninitialized pointers, causing **undefined behavior** and frequent crashes. Safe Rust **does not have null references**, encodes absence with `Option<T>`, and **prevents uninitialized references at compile time**. Any attempt to bypass this requires `unsafe` and raw pointers, which explicitly opt out of guarantees and should be minimized and audited.

Security Implications
Null pointer dereferences often cause denial-of-service crashes and can mask deeper memory safety issues. Rust's design eliminates this in safe code by construction: you must either handle absence (`Option`) or you cannot express the operation at all. This reduces crash risk and eliminates a broad class of memory misuse.

Reproducibility Note
All example files from this section are available in the downloadable folder:
null_pointer_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. Uncomment the noted lines in the Rust examples to see **compile-time** errors; run the unwrap case to see the **panic**.

---

4. Out-of-Bounds Read/Write

Description
Out-of-bounds (OOB) errors occur when code reads from or writes to memory outside the valid range of an array or buffer. In C/C++, OOB access is **undefined behavior (UB)**: the program might appear to work, crash, or corrupt memory. In **safe Rust**, indexing performs **bounds checks**; out-of-range access **panics** at runtime, and some constant out-of-bounds cases can be rejected at **compile time**. Rust also provides safe APIs like `.get()` and iterators to **avoid panics** entirely.

Example 1: Out-of-Bounds READ

C++ Version – Reading Past the End

```cpp
// oob_example1_read.cpp
#include <iostream>
int main() {
    int arr[3] = {1, 2, 3};        // Line 3: valid indices 0..2
    std::cout << arr[5] << "\n";   // Line 4: DANGEROUS -- OOB READ (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** C++ does not insert runtime bounds checks for raw arrays; the standard treats OOB as **undefined behavior**.
- **What is dangerous:**
  - **Line 4** reads from `arr[5]` which is outside the object `arr[0..2]`. The read may return a “plausible” value, crash, or trigger memory corruption depending on layout and optimizations.

Rust Version – Bounds-Checked Read

```rust
// oob_example1_read.rs
fn main() {
    let arr = [1, 2, 3];           // Line 2: valid indices 0..2
    println!("{}", arr[5]);        // Line 3: runtime panic -- index out of bounds
}
```

Explanation (Rust):

- **How it's prevented:** Rust inserts a **bounds check** for `arr[5]`.
  - **Line 3** panics with a clear message (e.g., "index out of bounds: the len is 3 but the index is 5").
- **Non-panicking alternative:** Use `.get()` to handle the OOB case explicitly:

```
// oob_example1_read_safe.rs
fn main() {
    let arr = [1, 2, 3];
    match arr.get(5) {              // Line 3: SAFE -- returns Option<&i32>
        Some(v) => println!("{}", v),
        None => println!("Index out of bounds"),
    }
}
```

Example 2: Out-of-Bounds WRITE

C++ Version – Writing Past the End

```
// oob_example2_write.cpp
#include <iostream>
int main() {
    int arr[3] = {0, 0, 0};       // Line 3: valid indices 0..2
    arr[5] = 99;                  // Line 4: DANGEROUS -- OOB WRITE (UB)
    std::cout << arr[5] << "\n";  // Line 5: also OOB READ (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** No mandatory bounds checks for raw arrays.
- **What is dangerous:**
  - **Line 4** writes to memory past the array's end — can corrupt adjacent stack data, cause later crashes, or enable exploits.
  - **Line 5** reads from the same invalid slot — still **UB**.

Rust Version – Bounds-Checked Write

```
// oob_example2_write.rs
```

```rust
fn main() {
    let mut arr = [0; 3];          // Line 2: valid indices 0..2
    arr[5] = 99;                   // Line 3: runtime panic -- index out of bounds
    println!("{}", arr[5]);
}
```

Explanation (Rust):

- **How it's prevented:** The write at **Line 3** triggers the bounds check and **panics**, preventing memory corruption.

Example 3: Loop Overrun (Read Past End)

C++ Version – Loop Reads One Past End

```cpp
// oob_example3_loop_read.cpp
#include <iostream>
int main() {
    int arr[3] = {1, 2, 3};
    int sum = 0;
    for (int i = 0; i <= 3; ++i) {  // Line 5: DANGEROUS -- i==3 reads past end
        sum += arr[i];              // Line 6: UB when i==3
    }
    std::cout << sum << "\n";
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The compiler assumes loops stay within bounds unless proven otherwise; UB gives the optimizer freedom.
- **What is dangerous:**
    - **Line 5–6** read `arr[3]` (invalid). The result may vary across runs/builds.

Rust Version – Loop with Bounds Check (Panic) and Safe Iteration

```rust
// oob_example3_loop_read.rs
```

```
fn main() {
    let arr = [1, 2, 3];
    let mut sum = 0;
    for i in 0..=3 {              // Line 4: DANGEROUS -- i==3 is out of bounds
        sum += arr[i];            // Line 5: runtime panic when i==3
    }
    println!("{}", sum);
}
```

Safe alternative using iterators (no panic):

```
// oob_example3_loop_read_safe.rs
fn main() {
    let arr = [1, 2, 3];
    let sum: i32 = arr.iter().sum();  // SAFE -- iterators never go out of bounds
    println!("{}", sum);
}
```

Compile-Time Prevention (Constant Index)

```
// oob_example_const_compile.rs
fn main() {
    let arr = [0; 3];
    let x = arr[10];             // DANGEROUS -- constant OOB; can be rejected at compile time
    println!("{}", x);
}
```

Explanation (Rust):

- When the compiler can **prove** at compile time that the index is out of bounds (e.g., constant index on a known-size array), it will **reject** the program during compilation. Otherwise, safe indexing is still checked at runtime and **panics** on violation.

CWE Mapping

- **CWE-125:** Out-of-bounds Read
- **CWE-787:** Out-of-bounds Write

- **CWE-119:** Improper Restriction of Operations within the Bounds of a Memory Buffer

---

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
OOB read	Allowed; UB at **Ex1:L4**, **Ex3:L6**	Panic at **Ex1:L3**, **Ex3:L5**; `.get()` is safe
OOB write	Allowed; UB at **Ex2:L4**	Panic at **Ex2:L3**
Compile-time prevention	None for raw arrays	Constant OOB can be **compile error**
Safer iteration/access	Manual checks or safer containers	`.get()`, iterators avoid out-of-bounds by design

---

Developer Takeaway
C/C++ permit out-of-bounds array access on raw arrays, resulting in **undefined behavior** that can corrupt memory or crash unpredictably. Safe Rust **always checks** bounds for indexing and offers APIs (`.get()`, iterators) that **avoid panics** completely by making OOB explicit and non-fatal. Where possible, Rust can even **reject** constant out-of-bounds at compile time.

Security Implications
Out-of-bounds accesses are a core source of memory corruption and exploit primitives (e.g., info leaks via OOB reads, control flow hijacking via OOB writes). Rust prevents these in **safe code** through bounds checks and safer access patterns, effectively eliminating this class of vulnerability without sacrificing performance-critical code paths where `unsafe` is carefully encapsulated.

Reproducibility Note
All example files from this section are available in the downloadable folder:
out_of_bounds_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. For Rust, try both the **panic** examples and the **safe** alternatives (`.get()`, iterators).

---

5. Double Free

Description
A double free occurs when the same memory is deallocated **more than once**. In C/C++, this is **undefined behavior (UB)** and is often exploitable (e.g., heap metadata corruption, tcache poisoning). In **safe Rust**, ownership and move semantics ensure each allocation is dropped

exactly once; attempting to "free" a value twice results in a **compile-time error**. Shared ownership in Rust is handled by `Rc`/`Arc`, which free memory **once** when the last strong reference is dropped.

Example 1: Delete Twice

C++ Version – `delete` Twice

```
// df_example1_delete_twice.cpp
#include <iostream>
int main() {
    int* p = new int(7);   // Line 3: allocate
    delete p;               // Line 4: free once
    delete p;               // Line 5: DANGEROUS -- double delete (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The C++ language does not track whether `p` has already been freed; dereferencing or deleting again is **UB**.
- **What is dangerous:**
  - **Line 5** attempts to free already-freed memory; behavior ranges from silent corruption to immediate crash.

Rust Version – Single Drop Enforced

```
// df_example1_single_drop.rs
fn main() {
    let p = Box::new(7);
    drop(p);                 // Line 3: moves and drops; memory freed
exactly once
    // drop(p);              // Line 4: COMPILE ERROR -- use of moved value
`p` (prevents double free)
}
```

Explanation (Rust):

- **How it's prevented:** After `drop(p)`, ownership is consumed. Uncommenting **Line 4** yields a compile error ("use of moved value"), ensuring memory is freed **once**.

Example 2: Aliased Pointer, Double Delete

C++ Version – Two Raw Aliases, Both `delete`

```
// df_example2_alias_delete.cpp
#include <iostream>
int main() {
    int* p = new int(42);    // Line 3: allocate
    int* q = p;              // Line 4: alias same allocation
    delete p;                // Line 5: free
    delete q;                // Line 6: DANGEROUS -- double delete via
alias (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The compiler has no aliasing metadata for raw pointers; both `p` and `q` appear valid.
- **What is dangerous:**
  - **Line 6** frees already-released storage; outcome depends on allocator internals and can be exploitable.

Rust Version – Moves Prevent Aliased Free

```
// df_example2_move_alias.rs
fn main() {
    let p = Box::new(42);
    let q = p;              // Line 3: move; `p` is now invalid
    // drop(p);             // Line 4: COMPILE ERROR -- use of moved value
`p`
    drop(q);                // Line 5: OK: dropped exactly once
}
```

Explanation (Rust):

- **How it's prevented:** Only **one owner** exists at a time for `Box<T>`. Moving to `q` invalidates `p` at compile time, making double free impossible in safe code.

Example 3: `malloc`/`free` Twice (C) vs. Reference Counting (Rust)

C++ Version – `free` Twice

```cpp
// df_example3_malloc_free_twice.cpp
#include <cstdlib>
int main() {
    void* p = std::malloc(16); // Line 3: allocate
    std::free(p);              // Line 4: free
    std::free(p);              // Line 5: DANGEROUS -- double free (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The standard library won't stop you from freeing the same pointer twice; it's **UB** and often exploitable.

Rust Version – Shared Ownership Frees Once

```rust
// df_example3_rc_safe.rs
use std::rc::Rc;
fn main() {
    let a = Rc::new(String::from("hello"));
    let b = Rc::clone(&a);
    let c = Rc::clone(&a);
    // All clones share ownership; memory freed once when the last strong ref is dropped.
    println!("counts: strong={}, weak={}", Rc::strong_count(&a), Rc::weak_count(&a));
}
```

Explanation (Rust):

- **How it's prevented:** `Rc` uses reference counting; destruction occurs **exactly once** when the count hits zero. Aliasing is safe and double free cannot occur in safe Rust.

(Optional) Example 4: Reintroducing Double Free with `unsafe`

```
// df_example4_unsafe_raw_double_free.rs
// Demonstration only -- do not run.
fn main() {
    let b = Box::new(123);
    let raw = Box::into_raw(b);        // take ownership as raw pointer
    unsafe {
        // Convert back once and drop (frees memory):
        let _once = Box::from_raw(raw);
        // Convert back again and drop a second time -- DOUBLE FREE if
uncommented:
        // let _twice = Box::from_raw(raw);
    }
}
```

Explanation:

- Only by using **`unsafe`** and raw pointers can you subvert Rust's guarantees and simulate a double free. Such code is explicitly opt-in and should be minimized and audited.

CWE Mapping

- **CWE-415:** Double Free
- **CWE-416:** Use After Free
- **CWE-762:** Mismatched Memory Management Routines

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Double delete/free	Allowed; UB at **Ex1:L5**, **Ex2:L6**, **Ex3:L5**	Prevented by ownership; compile error at **Ex1:L4**, **Ex2:L4**
Aliased ownership	Raw pointers can alias freely	One owner for `Box<T>`; shared via `Rc`/`Arc` safely
Shared ownership	Manual, error-prone	`Rc`/`Arc` ref-counted; frees once on last drop
Boundary of safety	Not enforced by language	Safe Rust enforces; only bypassed with explicit `unsafe`

Developer Takeaway
In C/C++, double free results from manual memory management and lack of language-level ownership semantics. Safe Rust **prevents** double free at compile time via **moves** and **single-drop** semantics, and supports **safe shared ownership** via `Rc`/`Arc`. Double free can only be reintroduced in Rust with explicit `unsafe` and raw pointers, which isolates risk.

Security Implications
Double free is a common exploitation primitive on modern allocators (heap metadata corruption, UAF). Rust's model eliminates double free in **safe code**, significantly reducing attack surface.

Reproducibility Note
All example files from this section are available in the downloadable folder:
double_free_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. Uncomment the indicated lines in the Rust examples to observe **compile-time** prevention.

---

6. Memory Leaks

Description
A memory leak occurs when allocated memory is **never released** back to the system. In C/C++, leaks commonly arise from missing `delete`/`free`, pointer overwrites that lose track of the original allocation, or **reference cycles** (e.g., `std::shared_ptr`). In **safe Rust**, ordinary ownership drops free memory automatically at scope end (RAII). Leaks can still occur intentionally (e.g., `std::mem::forget`, `Box::leak`) or via **reference cycles** with `Rc`/`Arc` if `Weak` is not used to break cycles.

Example 1: Missing Deallocation

C++ Version – `new` Without `delete`

```
// leak_example1_no_delete.cpp
#include <iostream>
int main() {
    int* p = new int(42);   // Line 3: allocate
    // no delete -> memory leak
```

```
    std::cout << *p << "\n"; // Line 5: program ends without freeing `p`
    return 0;                // Leak persists until process exit
}
```

Explanation (C++):

- **Why it compiles:** C++ does not enforce freeing allocations; lifetime is manual.
- **What is dangerous:**
  - The allocation at **Line 3** is never released. Long-running processes accumulate leaks and can exhaust memory.

Rust Version – RAII Prevents Leaks by Default

```rust
// leak_example1_raii_drop.rs
fn main() {
    let p = Box::new(42);   // Line 2: allocation
    println!("{}", p);
    // Line 4: p is dropped automatically at end of scope -> no leak
}
```

Explanation (Rust):

- **Why it's safe:** Ownership ensures values are **dropped** when they go out of scope—memory is freed automatically.

Example 2: Pointer Overwrite Loses Original Allocation

C++ Version – Overwriting Pointer

```cpp
// leak_example2_overwrite_ptr.cpp
#include <iostream>
int main() {
    int* p = new int(1);     // Line 3: allocate block A
    p = new int(2);          // Line 4: DANGEROUS -- previous pointer lost; block A leaked
    std::cout << *p << "\n";
    delete p;                // Line 6: frees only block B; A still leaked
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** There's no automatic tracking to free the previous allocation when overwriting `p`.
- **What is dangerous:**
  - **Line 4** loses all references to the first allocation—**permanent leak**.

Rust Version – Moves and RAII Avoid This

- In Rust, you would create a **new** `Box` binding for the second allocation, and the first binding would be dropped automatically when it goes out of scope. Overwriting an owned value **drops the old value** (unless `mem::forget` is used).

Example 3: Reference Cycles

C++ Version – `std::shared_ptr` Cycle

```cpp
// leak_example3_shared_ptr_cycle.cpp
#include <memory>
#include <iostream>

struct Node {
    std::shared_ptr<Node> next; // Line 5: owning strong ref creates
potential cycle
    ~Node() { std::cout << "~Node\n"; }
};

int main() {
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();
    a->next = b;              // Line 12
    b->next = a;              // Line 13: cycle: a -> b -> a
    return 0;                 // Destructors do not run; refcounts never
reach zero -> leak
}
```

Explanation (C++):

- **Why it leaks:** Strong reference cycles keep counts > 0 even when variables go out of scope.

Fixed C++ – Break Cycle with `weak_ptr`

```
// leak_example3_shared_ptr_cycle_fixed.cpp
#include <memory>
#include <iostream>

struct Node {
    std::weak_ptr<Node> next;  // Line 5: use weak_ptr to break cycle
    ~Node() { std::cout << "~Node\n"; }
};

int main() {
    auto a = std::make_shared<Node>();
    auto b = std::make_shared<Node>();
    a->next = b;                 // Line 12: store weak ref
    b->next = a;                 // Line 13
    return 0;                    // Destructors run; no leak
}
```

Rust Version – `Rc` Cycle (Leak) vs. `Weak` (No Leak)

```
// leak_example2_rc_cycle.rs
use std::cell::RefCell;
use std::rc::Rc;

#[derive(Debug)]
struct Node {
    value: i32,
    next: RefCell<Option<Rc<Node>>>,
}

fn main() {
    let a = Rc::new(Node { value: 1, next: RefCell::new(None) });
    let b = Rc::new(Node { value: 2, next: RefCell::new(Some(a.clone()))
});
    *a.next.borrow_mut() = Some(b.clone()); // Line 15: create cycle a -> b
-> a
    println!("strong_count a={}, b={}", Rc::strong_count(&a),
Rc::strong_count(&b));
    // Leak: counts never drop to zero at end of main.
}
```

```rust
// leak_example3_rc_weak_break.rs
use std::cell::RefCell;
use std::rc::{Rc, Weak};

#[derive(Debug)]
struct Node {
    value: i32,
    next: RefCell<Option<Weak<Node>>>, // Line 8: Weak breaks cycles
}

fn main() {
    let a = Rc::new(Node { value: 1, next: RefCell::new(None) });
    let b = Rc::new(Node { value: 2, next:
RefCell::new(Some(Rc::downgrade(&a))) });
    *a.next.borrow_mut() = Some(Rc::downgrade(&b)); // Line 14: a ->(Weak)
b, b ->(Weak) a
    println!("strong_count a={}, b={}", Rc::strong_count(&a),
Rc::strong_count(&b));
    // No leak: strong counts reach zero at end of main.
}
```

Example 4: Explicit/Intentional Leaks in Rust

```rust
// leak_example4_mem_forget.rs
fn main() {
    let s = String::from("leak me");
    std::mem::forget(s); // Line 3: intentionally leak by forgetting to
drop
    // Alternative: Box::leak to obtain a 'static reference that is never
freed.
}
```

Explanation:

- Rust can leak by design when you explicitly opt out of dropping (e.g., `mem::forget`, `Box::leak`). This is rare and deliberate.

CWE Mapping

- **CWE-401:** [Missing Release of Memory after Effective Lifetime]
- CWE-772: [Missing Release of Resource after Effective Lifetime]
- CWE-404: [Improper Resource Shutdown or Release]

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Missing deallocation	Allowed; leak at **Ex1 (C++)**	Dropped automatically at scope end (RAII)
Pointer overwrite	Allowed; leak at **Ex2:L4**	Overwrite drops old value automatically (unless leaked)
Reference cycles	`shared_ptr` strong cycles leak	`Rc` strong cycles leak; fix with `Weak`
Intentional leaks	Possible (e.g., never calling `delete`)	Possible via `mem::forget` / `Box::leak` (explicit, opt-in)

Developer Takeaway

C/C++ leak risks come from **manual memory management** and **reference cycles**. Safe Rust uses ownership and RAII to **free memory automatically**; leaks mainly arise from **`Rc`/`Arc` cycles** or **explicitly opting out** of drop. Break cycles with `Weak`, and avoid `mem::forget`/`Box::leak` unless you truly need them.

Security Implications

Leaks degrade reliability and can lead to **resource exhaustion** (DoS). While less directly exploitable than UAF/overflow, unmanaged leaks in long-running services are high-risk. Rust's defaults significantly reduce accidental leaks and make remaining cases explicit and auditable.

Reproducibility Note

All example files from this section are available in the downloadable folder:
memory_leak_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.

3. Observe how Rust drops values automatically; try the `Rc` cycle vs. `Weak` fix to see the difference.

---

7. Race Condition (Data Race)

Description
A **data race** occurs when two or more threads access the same memory location **concurrently**, at least one access is a **write**, and there is **no synchronization** to order the accesses. In C/C++, such races are **undefined behavior (UB)** and can yield corrupted state or nondeterministic crashes. In **safe Rust**, the type system and marker traits (`Send`, `Sync`) prevent sharing unsafely by default; you must use safe primitives like `Arc<Mutex<_>>` or **atomics** for shared mutable state. Rust can also reject race-prone code at **compile time**.

Example 1: Increment Without Synchronization

C++ Version – Shared Counter Without Mutex

```cpp
// race_example1_no_mutex.cpp
#include <iostream>
#include <thread>
#include <vector>

int main() {
    int counter = 0;                          // Line 6: shared non-atomic
    const int threads = 8;
    const int iters = 100000;
    std::vector<std::thread> ts;

    for (int t = 0; t < threads; ++t) {
        ts.emplace_back([&]() {
            for (int i = 0; i < iters; ++i) {
                counter++;                    // Line 14: DANGEROUS -- data race (UB)
            }
        });
    }
    for (auto& th : ts) th.join();
    std::cout << "counter=" << counter << "\n"; // Line 20: nondeterministic result
    return 0;
```

```
}
```

Explanation (C++):

- **Why it compiles:** The compiler does not insert synchronization for raw integer increments. Multiple threads interleave unsafely.
- **What is dangerous:**
    - **Line 14** performs a read-modify-write without **atomicity** or a **mutex**, causing races. The final count is < `8 × 100000` unpredictably.

Rust Version – Compile-Time Prevention and Safe Fix

```
// race_example1_compile_error.rs
// This example shows a compile-time error if you try to capture &mut
across threads.
fn main() {
    let mut counter = 0;                        // Line 3: not shareable as
&mut across threads
    let mut handles = Vec::new();
    for _ in 0..8 {
        // Uncommenting inside will cause a compile error: cannot capture
`counter` by reference
        handles.push(std::thread::spawn(|| {
            for _ in 0..100_000 {
                // counter += 1; // COMPILE ERROR if enabled
            }
        }));
    }
    for h in handles { h.join().unwrap(); }
    println!("{}", counter);
}
```

Explanation (Rust):

- **How it's prevented:** The compiler rejects sharing a mutable, non-`Send/Sync` reference across threads. Safe Rust **forces** synchronization primitives.

Safe Fix – `Arc<Mutex<_>>`

```
// race_example2_arc_mutex.rs
```

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));     // Line 5: shared, synchronized state
    let mut handles = vec![];
    for _ in 0..8 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..100_000 {
                let mut guard = c.lock().unwrap(); // Line 11: lock before mutate
                *guard += 1;
            }
        }));
    }
    for h in handles { h.join().unwrap(); }
    println!("counter={}", *counter.lock().unwrap()); // Deterministic: 800000
}
```

Alternative – Lock-Free Atomic

```rust
// race_example3_atomic.rs
use std::sync::atomic::{AtomicUsize, Ordering};
use std::sync::Arc;
use std::thread;

fn main() {
    let counter = Arc::new(AtomicUsize::new(0));      // Line 6: atomic shared counter
    let mut handles = vec![];
    for _ in 0..8 {
        let c = Arc::clone(&counter);
        handles.push(thread::spawn(move || {
            for _ in 0..100_000 {
                c.fetch_add(1, Ordering::Relaxed);   // Line 12: atomic increment
            }
        }));
```

```
    }
    for h in handles { h.join().unwrap(); }
    println!("counter={}", counter.load(Ordering::Relaxed)); //
Deterministic: 800000
}
```

Example 2: Unsynchronized Read/Write Flag

C++ Version – Spin on a Non-Atomic Flag

```
// race_example2_read_write_flag.cpp
#include <iostream>
#include <thread>
#include <chrono>

bool running = true;                            // Line 6: shared non-atomic
flag

void worker() {
    size_t spins = 0;
    while (running) {                           // Line 10: read race with main
writer
        ++spins;
    }
    std::cout << "worker spins=" << spins << "\n";
}

int main() {
    std::thread th(worker);
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    running = false;                            // Line 19: write without
synchronization
    th.join();
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The compiler can cache `running` in a register; the worker might **never see the update** (reordering, tearing).

- **Fix:** Use `std::atomic<bool>` and appropriate memory ordering, or a mutex/condition variable.

Rust Equivalent – Requires Atomics

- In Rust, to share a stop flag, use `Arc<AtomicBool>` or `Arc<Mutex<bool>>`. The compiler won't let you share `&mut bool` across threads without synchronization.

(Optional) Example 3: Explicit `unsafe` Data Race in Rust

```
// race_example4_unsafe_raw.rs
// DO NOT RUN -- Demonstrates that a data race requires explicit `unsafe`
in Rust.
static mut COUNTER: i32 = 0;

fn main() {
    let mut handles = vec![];
    for _ in 0..8 {
        handles.push(std::thread::spawn(|| {
            for _ in 0..100_000 {
                unsafe { COUNTER += 1; } // DANGEROUS: unsynchronized RMW
of shared static -> UB
            }
        }));
    }
    for h in handles { h.join().unwrap(); }
    unsafe { println!("COUNTER={}", COUNTER); }
}
```

Explanation:

- Safe Rust disallows data races by construction; reintroducing one requires `unsafe` or FFI. This isolates risk and forces explicit acknowledgment.

CWE Mapping

- **CWE-362:** Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')
- **CWE-366:** Race Condition within a Thread
- **CWE-664:** Improper Control of a Resource Through its Lifetime

- **CWE-667:** Improper Locking

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Shared mutable state	Allowed; races are UB at **Ex1:L14**, **Ex2:L10/19**	Compile-time rejection without `Arc<Mutex<_>>` or atomics
Determinism/correctness	Nondeterministic results, stale reads	Deterministic with `Mutex`; correct with `Atomic*`
Compile-time safety	None	Enforced: borrow checker, `Send`/`Sync` trait bounds
Opting out	N/A (always possible)	Only with explicit `unsafe` (e.g., static mut / raw pointers)

Developer Takeaway
C/C++ permit unsynchronized concurrent access, leading to **undefined behavior** and fragile code. Safe Rust prevents data races at **compile time**; to share mutable state you must use `Arc<Mutex<_>>` or **atomics**, which encode the synchronization in the types. This makes concurrency correct by default and pushes unsafe patterns into explicit `unsafe` blocks.

Security Implications
Races are notoriously hard to reproduce and exploit, but they can lead to integrity violations, information leaks, and bypasses of security checks. Rust's model drastically reduces this class in safe code by requiring explicit, type-checked synchronization.

Reproducibility Note
All example files from this section are available in the downloadable folder:
race_condition_section_pkg.zip — Download

To replicate:
1. Install **g++ (C++17+, with `-pthread`)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. Observe that unsafe patterns in Rust are confined to the `unsafe` demo; the safe versions compile and run deterministically.

---

8. Uninitialized Memory

Description
 Uninitialized memory bugs occur when a program **reads** from variables or buffers that were **never given a defined value**. In C/C++, this is **undefined behavior (UB)** and may yield garbage data, crashes, or subtle logic errors. **Safe Rust** prohibits using a variable before it is initialized and enforces definite initialization at compile time. For low-level cases, Rust provides `std::mem::MaybeUninit<T>` to initialize data manually **without creating UB**, as long as you don't read before fully writing.

Example 1: Uninitialized Local

C++ Version – Reading an Uninitialized Stack Variable

```
// uninit_example1_stack_var.cpp
#include <iostream>
int main() {
    int x;                          // Line 3: uninitialized local
(indeterminate value)
    std::cout << x << "\n";         // Line 4: DANGEROUS -- use of
uninitialized variable (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The language allows locals without initializers; reading them is **UB**.
- **What is dangerous:**
    - **Line 4** reads an **indeterminate** value. On some runs it may "work," masking the bug.

Rust Version – Compile-Time Prevention

```
// uninit_example1_compile_error.rs
fn main() {
    let x: i32;                     // Line 2: declared but not initialized
    // println!("{}", x);           // Line 3: COMPILE ERROR -- use of
possibly-uninitialized `x`
}
```

Explanation (Rust):

- **How it’s prevented:** The compiler enforces **definite assignment**; any read of an uninitialized binding is rejected at compile time.

Example 2: Uninitialized Array vs. Initialized Array

C++ Version – Reading an Uninitialized Array

```
// uninit_example2_uninit_array.cpp
#include <iostream>
int main() {
    int arr[3];                 // Line 3: uninitialized array
    int sum = 0;
    for (int i = 0; i < 3; ++i) {
        sum += arr[i];          // Line 6: DANGEROUS -- reading
indeterminate values (UB)
    }
    std::cout << sum << "\n";
    return 0;
}
```

Rust Version – Initialized Array and Safe Iteration

```
// uninit_example3_array_default.rs
fn main() {
    let arr = [0i32; 3];        // Line 2: fully initialized array
    let sum: i32 = arr.iter().sum();
    println!("{}", sum);
}
```

Explanation:

- Rust arrays must be **fully initialized** before use. Iterators ensure safe access.

Example 3: Heap Allocation Without Initialization

C++ Version – `new int` Without Initializer

```
// uninit_example3_new_no_init.cpp
```

```cpp
#include <iostream>
int main() {
    int* p = new int;            // Line 3: uninitialized heap allocation
    std::cout << *p << "\n";     // Line 4: DANGEROUS -- reading
uninitialized memory (UB)
    delete p;
    return 0;
}
```

Rust Version – Initialize Before Use

```rust
// uninit_example2_init_before_use.rs
fn main() {
    let x: i32;
    x = 10;                      // Line 3: initialized before use
    println!("{}", x);           // Line 4: OK
}
```

Example 4: Low-Level Control with `MaybeUninit`

Safe Pattern (Write-All-Before-Read)

```rust
// uninit_example4_maybeuninit.rs
use std::mem::MaybeUninit;
fn main() {
    // SAFE pattern: create uninitialized backing storage, then write every
element.
    let mut buf: [MaybeUninit<u32>; 3] = unsafe {
MaybeUninit::uninit().assume_init() };
    for i in 0..3 {
        buf[i] = MaybeUninit::new(i as u32 + 1);   // initialize each
element
    }
    // Now transmute to a fully-initialized array. This is safe because
we've written all elements.
    let arr: [u32; 3] = unsafe { std::mem::transmute(buf) };
    println!("{:?}", arr);
}
```

Danger Pattern (Don't Do This)

```
// uninit_example5_maybeuninit_UB.rs
// Demonstration only -- do NOT run. UB if you read before fully
initializing.
use std::mem::MaybeUninit;
fn main() {
    let buf: [MaybeUninit<u32>; 3] = unsafe {
MaybeUninit::uninit().assume_init() };
    // println!("{:?}", unsafe { std::mem::transmute::<_, [u32; 3]>(buf)
}); // UB: reading uninitialized memory
}
```

Explanation:

- `MaybeUninit<T>` allows **uninitialized storage** without immediate UB, but reading before writing all bytes of every element is **undefined behavior**. The safe pattern is: **allocate** → **write all** → **read**.

CWE Mapping

- **CWE-457:** [Use of Uninitialized Variable]
- **CWE-908:** [Use of Uninitialized Resource]
- **CWE-824:** [Access of Uninitialized Pointer]

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Use before initialize (local/heap)	Allowed; UB at **Ex1:L4**, **Ex3:L4**	Compile-time error at **Ex1 (Rust):L3**; must init first
Uninitialized arrays	Allowed; UB at **Ex2:L6**	Arrays must be fully initialized before use
Low-level uninitialized handling	Manual, error-prone	`MaybeUninit` provides explicit, auditable initialization
Boundary of safety	Not enforced by language	Enforced in safe Rust; UB only via misuse/`unsafe`

Developer Takeaway
C/C++ let you accidentally read **indeterminate** data, producing UB and fragile behavior. Safe Rust prohibits use-before-initialize at **compile time**, and when you need low-level control, `MaybeUninit` gives you a safe pattern—**write all bytes before any read**—so you don't slip into UB.

Security Implications
Uninitialized reads can leak sensitive data (leftover stack/heap contents) or cause logic corruption. Rust's definite-initialization and safe APIs shut this down in safe code, making such bugs rare and auditable when low-level patterns are required.

Reproducibility Note
All example files from this section are available in the downloadable folder:
uninitialized_memory_section_pkg.zip — Download
To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. For Rust, try the compile-error case by uncommenting the noted line, and compare with the safe `MaybeUninit` pattern.

---

9. Dangling Pointers

Description
A **dangling pointer** refers to memory that has been **freed or gone out of scope**, but the pointer/reference still exists and is later used. In C/C++, this is **undefined behavior (UB)** and can lead to crashes, silent corruption, or exploitable conditions. In **safe Rust**, lifetimes and the borrow checker **prevent references from outliving the data** they refer to, so dangling references cannot be expressed. You can only reintroduce this risk via **`unsafe`** raw pointers.

Example 1: Returning the Address of a Local

C++ Version – Dangling on Return

```
// dangling_example1_return_local.cpp
#include <iostream>
int* make_ptr() {
    int x = 123;                    // Line 2: stack local
    return &x;                      // Line 3: DANGEROUS -- returns address of
```

```
dead stack object
}
int main() {
    int* p = make_ptr();          // Line 6: `p` now dangles
    std::cout << *p << "\n";      // Line 7: use of dangling pointer (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** Raw pointers carry no lifetime information; the compiler can't prove &x escapes its scope.
- **What is dangerous:**
  - **Line 3** returns a pointer to a stack variable destroyed on return.
  - **Line 7** dereferences a dangling pointer → **UB** (may crash or print garbage).

Rust Version – Borrow Checker Rejects It

```
// dangling_example1_borrow_checker.rs
// Returning a reference to a local is rejected at compile time.
fn make_ref<'a>() -> &'a i32 {
    let x = 123;                  // Line 2: local
    // &x                         // If returned, COMPILE ERROR -- `x` does
not live long enough
    unimplemented!()
}
fn main() { let _ = 0; }
```

Explanation (Rust):

- **How it's prevented:** The **lifetime** `'a` on the return type must outlive x, which it cannot. The compiler rejects the code if you try to return &x.

Example 2: Iterator/Buffer Invalidated by Reallocation

C++ Version – `std::vector` Reallocation

```
// dangling_example2_vector_realloc.cpp
```

```
#include <iostream>
#include <vector>
int main() {
    std::vector<int> v = {1, 2, 3};
    int* p = &v[0];                  // Line 5: pointer into vector's buffer
    v.push_back(4);                  // Line 6: may reallocate and move
buffer
    std::cout << *p << "\n";         // Line 7: DANGEROUS -- `p` may now
dangle (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** Raw pointer p is not invalidated at compile time; push_back may reallocate and move elements, invalidating p.
- **What is dangerous:**
    - **Line 6–7**: If reallocation occurs, dereferencing p is **UB**.

Rust Version – Borrow Then Mutate is Rejected; Safe Alternatives

```
// dangling_example2_vec_borrow_then_push.rs
fn main() {
    let mut v = vec![1, 2, 3];
    let r = &v[0];                  // Line 3: immutable borrow of element
    // v.push(4);                   // Line 4: COMPILE ERROR -- cannot
borrow `v` as mutable while `r` is live
    println!("{}", r);              // Line 5: safe; borrow ends here
}
```

```
// dangling_example3_index_or_clone.rs
fn main() {
    let mut v = vec![1, 2, 3];
    let x = v[0];                   // Line 3: copy the value (i32: Copy)
    v.push(4);                      // Line 4: mutate after the copy; no
outstanding borrow
    println!("{}", x);              // Line 5: safe
}
```

Explanation (Rust):

- **How it's prevented:** Rust will **not** allow a mutation that could reallocate while an immutable borrow exists. Copying the value or using indices avoids dangling.

Example 3: Free, Save Pointer, Use Later

C++ Version – Free Then Later Use

```cpp
// dangling_example3_free_then_later_use.cpp
#include <iostream>
void free_then_keep(int** out) {
    *out = new int(7);              // Line 2: allocate
    delete *out;                    // Line 3: free
    // returning with *out still pointing to freed memory
}
int main() {
    int* p = nullptr;
    free_then_keep(&p);             // Line 9: p is dangling immediately
    // later...
    std::cout << *p << "\n";        // Line 11: use of dangling pointer (UB)
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** The type system doesn't track that `*out` became invalid after `delete`.
- **What is dangerous:**
    - **Line 11** dereferences a pointer to freed memory → **UB**.

Rust Note:

- In safe Rust, after `drop(x)` (or end of scope), there is **no valid reference** left to the memory; the compiler prevents uses that would outlive the owner.

(Optional) Example 4: Reintroducing Dangling with `unsafe`

```rust
// dangling_example4_unsafe_raw.rs
// Demonstration only -- using raw pointers can create dangling references.
```

```
DO NOT RUN.
fn main() {
    let mut v = vec![1, 2, 3];
    let p: *const i32 = &v[0];        // raw pointer into vector buffer
    v.reserve(1000);                   // may reallocate
    unsafe {
        // println!("{}", *p);        // DANGEROUS -- `p` may dangle after
reallocation (UB)
    }
}
```

Explanation:

- Only **`unsafe`** raw pointers can bypass Rust's lifetime checks and reproduce dangling-like bugs; this is explicitly opt-in and should be audited.

CWE Mapping

- **CWE-825:** Expired Pointer Dereference (Dangling Pointer)
- (Related) **CWE-416:** Use After Free

Analysis and Comparison

Feature	C++ Behavior (lines)	Rust Behavior (lines)
Return address of local	Allowed; UB at **Ex1:L7**	Rejected by borrow checker at **Ex1 (Rust)**
Reallocation invalidates refs	Allowed; UB at **Ex2:L7**	Mutation blocked while borrow alive; safe copy alternative
Free then use later	Allowed; UB at **Ex3:L11**	Owner drop ends all borrows; cannot outlive owner
Bypassing safety	Always possible	Only via explicit `unsafe` raw pointers

---

Developer Takeaway
C/C++ allow **dangling pointers** because raw pointers have no lifetime tracking. Rust's lifetimes and borrow rules **encode validity in the type system**, rejecting patterns that would outlive the

data (returning refs to locals, mutating while borrowed, etc.). Only by opting into `unsafe` can you subvert these guarantees.

Security Implications
Dangling pointers are a frequent source of **use-after-free** and memory corruption. Rust's model prevents them in **safe code**, shrinking the attack surface and improving reliability.

Reproducibility Note
All example files from this section are available in the downloadable folder:
dangling_pointers_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. For Rust, uncomment the indicated lines to observe **compile-time** rejections; the unsafe demo is provided for contrast and should **not** be run.

---

10. Integer Overflow / Underflow

Description
Integer overflow (or underflow) happens when an arithmetic operation exceeds the representable range of its type. In **C/C++**, **signed** overflow is **undefined behavior (UB)**, while **unsigned** overflow wraps modulo 2^n. In **Rust**, debug builds **panic** on overflow; release builds **wrap** by default, but Rust provides explicit, safe APIs (`checked_*`, `saturating_*`, `wrapping_*`) and can catch overflow **at compile time** in certain **const** contexts. This makes overflow behavior **intentional and explicit**.

Example 1: Signed Overflow (UB) vs. Rust Debug Panic

C++ Version – Signed Overflow (Undefined Behavior)

```
// int_overflow_example1_signed_ub.cpp
#include <iostream>
#include <climits>
int main() {
    int a = INT_MAX;            // Line 4: 2147483647 on 32-bit int
    int b = 1;
    int c = a + b;              // Line 6: DANGEROUS -- signed overflow is
UB
```

```
    std::cout << "c=" << c << "\n"; // May print a negative value or be optimized unpredictably
    return 0;
}
```

Explanation (C++):

- **Why it compiles:** C++ permits the operation, but **signed overflow is UB**; the optimizer may assume it never happens and transform code accordingly.
- **What is dangerous:**
    - **Line 6** invokes UB; results vary (wrap-like values, weird behavior, or miscompilations).

Rust Version – Debug Panic

```
// int_overflow_example1_debug_panic.rs
fn main() {
    let a: i32 = i32::MAX;       // Line 2
    let b: i32 = 1;
    let _c = a + b;              // Line 4: debug builds panic; release wraps by default
    println!("done");
}
```

Explanation (Rust):

- **Debug build:** overflow **panics** with a clear message.
- **Release build:** overflow **wraps** (two's complement), unless you choose an explicit checked/saturating/wrapping API below.

Example 2: Explicitly Safe Arithmetic in Rust

Checked Arithmetic (No UB, No Wrap Surprises)

```
// int_overflow_example2_checked.rs
fn main() {
    let a: i32 = i32::MAX;
    let b: i32 = 1;
    match a.checked_add(b) {     // Line 4: SAFE -- returns Option<i32>
```

```
        Some(v) => println!("sum={}", v),
        None => println!("overflow detected"),
    }
}
```

Saturating Arithmetic (Clamp to Bounds)

```
// int_overflow_example3_saturating.rs
fn main() {
    let a: i32 = i32::MAX;
    let b: i32 = 1;
    let v = a.saturating_add(b); // Line 4: saturates at i32::MAX
    println!("saturating sum={}", v);
}
```

Wrapping Arithmetic (Intentional Wrap)

```
// int_overflow_example4_wrapping.rs
fn main() {
    let a: u32 = u32::MAX;
    let v = a.wrapping_add(1); // Line 3: wraps to 0 explicitly
    println!("wrap={}", v);
}
```

Example 3: Widen-Then-Multiply (Avoid Overflow)

C++ Version – Promote First

```
// int_overflow_example3_multiplies.cpp
#include <iostream>
#include <climits>
int main() {
    int x = 50000;               // Line 4
    int y = 50000;               // Line 5
    long long p = (long long)x * y; // Line 6: safe: promote before multiply
    int q = x * y;               // Line 7: DANGEROUS -- signed overflow (UB)
```

```
    std::cout << "p=" << p << " q=" << q << "\n";
    return 0;
}
```

Rust Version – Widen Before Multiply

```
// int_overflow_example6_widen_then_mul.rs
fn main() {
    let x: i32 = 50_000;
    let y: i32 = 50_000;
    let p: i64 = (x as i64) * (y as i64); // Line 4: safe widen-then-multiply
    // let q: i32 = x * y;               // Line 5: in debug, panic; in release, wrap (avoid)
    println!("p={}", p);
}
```

Compile-Time Protection in Const Contexts (Rust)

```
// int_overflow_example5_const_compile_error.rs
// Demonstrates compile-time overflow detection in a const context.
const _: u8 = 255 + 1; // Line 2: COMPILE ERROR -- attempt to compute `256_u8` which overflows
fn main() {}
```

Explanation:

- Rust **rejects** overflows in `const` **evaluation**, providing true **compile-time** protection when values are fully known at compile time.

CWE Mapping

- **CWE-190:** [Integer Overflow or Wraparound]()
- (Related) **CWE-191:** [Integer Underflow (Wraparound)]()

Analysis and Comparison

Feature	C/C++ Behavior (lines)	Rust Behavior (lines)
Signed overflow	UB at **Ex1:L6**, **Ex3:L7**	Debug: panic; Release: wrap (use explicit APIs)
Unsigned overflow	Well-defined wrap	Use `wrapping_*` for explicit intent
Compile-time protection	Limited (depends on tools, not language)	`const` contexts: **compile-time error** on overflow
Safer arithmetic options	Manual checks / wider types	`checked_*`, `saturating_*`, `wrapping_*`, widen as needed

---

Developer Takeaway
C/C++ **signed overflow is UB** and unsigned silently wraps, which can produce brittle, exploitable edge cases. Rust makes overflow **visible and controllable**: you can get panics in debug, explicit checked/saturating/wrapping operations in production, and even **compile-time** errors in constant expressions. Prefer **checked** math for untrusted inputs, or **widen** types before heavy arithmetic.

Security Implications
Integer overflows fuel **buffer mis-sizing, allocation truncation, and pointer arithmetic mistakes**, often leading to **out-of-bounds** or **heap corruption**. Rust's explicit arithmetic modes and const-time checks reduce the chance of overflow-driven memory bugs in safe code.

Reproducibility Note
All example files from this section are available in the downloadable folder:
integer_overflow_section_pkg.zip — Download

To replicate:

1. Install **g++ (C++17+)** and **Rust (rustc)**.
2. Unzip the folder and follow `README.txt` for compile/run commands.
3. Try debug vs. release in Rust, and compare **checked/saturating/wrapping** results and the **const compile-time** error.

Acknowledgment of Tool Usage

The author utilized OpenAI's ChatGPT model as a supplementary tool for generating initial code examples, refining explanatory text, and structuring sections of this report. All outputs from the model were independently reviewed, tested, and adapted by the author to ensure technical accuracy, contextual relevance, and alignment with the study's objectives.

References

American National Standards Institute. (2018). *ISO/IEC 14882:2017 Information technology — Programming languages — C++*. International Organization for Standardization. https://isocpp.org/std/the-standard

Cppreference.com. (n.d.). *C++ reference*. Retrieved August 8, 2025, from https://en.cppreference.com/w/

MITRE. (2025). *Common Weakness Enumeration (CWE) list*. https://cwe.mitre.org/

MITRE. (2025). *CWE-787: Out-of-bounds write*. https://cwe.mitre.org/data/definitions/787.html

MITRE. (2025). *CWE-120: Buffer copy without checking size of input*. https://cwe.mitre.org/data/definitions/120.html

MITRE. (2025). *CWE-416: Use after free*. https://cwe.mitre.org/data/definitions/416.html

MITRE. (2025). *CWE-476: Null pointer dereference*. https://cwe.mitre.org/data/definitions/476.html

MITRE. (2025). *CWE-362: Concurrent execution using shared resource with improper synchronization ('Race condition')*. https://cwe.mitre.org/data/definitions/362.html

MITRE. (2025). *CWE-401: Memory leak*. https://cwe.mitre.org/data/definitions/401.html

MITRE. (2025). *CWE-665: Improper initialization*. https://cwe.mitre.org/data/definitions/665.html

MITRE. (2025). *CWE-415: Double free*. https://cwe.mitre.org/data/definitions/415.html

MITRE. (2025). *CWE-825: Expired pointer dereference*. https://cwe.mitre.org/data/definitions/825.html

National Institute of Standards and Technology. (2025). *National Vulnerability Database (NVD)*. https://nvd.nist.gov/

Niko Matsakis, & Carol Nichols. (2023). *The Rust programming language*. No Starch Press. https://doc.rust-lang.org/book/

Rust Project Developers. (2025). *The Rust reference*. https://doc.rust-lang.org/reference/

Rust Project Developers. (2025). *The Rustonomicon*. https://doc.rust-lang.org/nomicon/

Rust Project Developers. (2025). *Rust unsafe code guidelines*. https://rust-lang.github.io/unsafe-code-guidelines/

The Rust Team. (2025). *Rust documentation*. https://www.rust-lang.org/

White, J., Holmes, D., & Matthews, A. (2017). *Safe systems programming in Rust*. *Communications of the ACM, 60*(10), 67–75. https://doi.org/10.1145/3123730

OpenAI. (2025, August 10). *ChatGPT* (GPT-5) [Large language model]. https://chat.openai.com/

Appendix A: Vulnerability-to-CWE Mapping and Rust Safety Features

Vulnerability Type	Relevant CWE(s)	Key Rust Safety Features Preventing It
Buffer Overflow	CWE-787, CWE-120, CWE-119	Runtime bounds checking on arrays/slices; `.get()` for safe optional access; compile-time constant index checks
Use-After-Free	CWE-416, CWE-415, CWE-562	Ownership model prevents access after move; borrow checker enforces valid lifetimes; no manual `free()` in safe code
Null Pointer Dereference	CWE-476, CWE-457, CWE-825	No null references in safe Rust (`Option<T>` used instead); borrow checker; compile-time guarantees
Out-of-Bounds Read	CWE-125, CWE-787	Automatic bounds checks; `.get()` for non-panicking access; slices/arrays safe indexing
Double Free	CWE-415, CWE-416, CWE-762	Ownership system ensures only one drop per value; move semantics prevent aliasing of freed objects
Memory Leak	CWE-401	Automatic memory management via ownership and `Drop`; RAII ensures resources freed at scope end
Race Condition (Data Race)	CWE-362, CWE-667, CWE-366, CWE-664	Compile-time enforcement of `Send`/`Sync`; `Mutex`, `RwLock`, `Arc` ensure safe concurrent access

Uninitialized Memory	CWE-457, CWE-908	Variables must be initialized before use; `MaybeUninit` API requires unsafe for partial init
Dangling Pointer	CWE-825, CWE-416	Borrow checker prevents use after free; lifetimes ensure references remain valid
Integer Overflow/Underflow	CWE-190, CWE-191	Checked arithmetic in debug mode; explicit `checked_*`, `wrapping_*`, `saturating_*` methods for intentional behavior