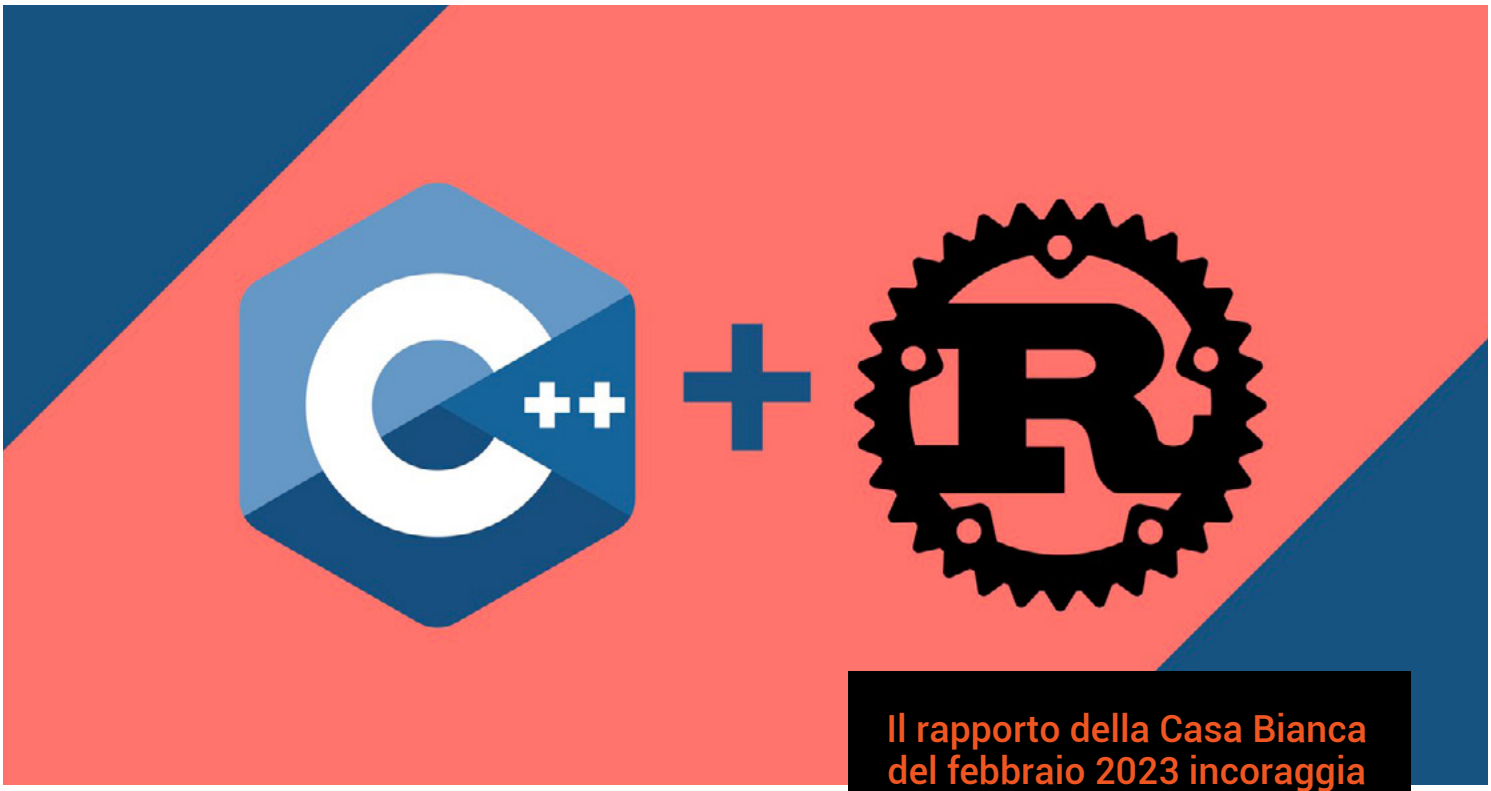
# RUST, SOLUZIONE ALLE VULNERABILITÀ DEL C++?



*A cura di Massimo Nannini (*)*

Il rapporto della Casa Bianca del febbraio 2023 incoraggia l'adozione di linguaggi di programmazione sicuri per mitigare le vulnerabilità di memoria, indicando Rust come un'alternativa efficace a C/C++. Rust offre una gestione sicura della memoria tramite un sistema di proprietà e verifiche a tempo di compilazione, eliminando molti errori comuni. Tuttavia, C++ resta preferito per le sue prestazioni e ottimizzazione delle risorse, nonostante i rischi legati alla sicurezza

**Sommario**

## Il report

Un report pubblicato nel Febbraio di quest'anno dalla Casa Bianca, mostra come il governo degli Stati Uniti abbia invitato gli sviluppatori di software ad abbandonare i linguaggi di programmazione che causano buffer overflow e altre vulnerabilità correlate all'uso della memoria.
Il documento esplora l'attuale approccio reattivo alla sicurezza informatica e il potenziale onere per gli utenti. In sostanza, sottolinea la necessità di adottare approcci proattivi alla sicurezza informatica, concentrandosi principalmente sull'eliminazione di intere classi di vulnerabilità piuttosto che semplicemente sulla correzione di quelle note.
L'amministrazione ha introdotto l'Executive Order 14028 insieme alla National Cybersecurity Strategy, che propugna il riequilibrio delle responsabilità e la priorità degli investimenti a lungo termine verso la sicurezza informatica, evidenziando l'importanza della cooperazione tra governo, industria e comunità tecnica nel raggiungimento di questi obiettivi.
Il documento sottolinea l'importanza di adottare misure proattive per eliminare interi gruppi di vulnerabilità software esistenti. In particolare, suggerisce che i produttori dovrebbero pubblicare i dati CVE (Common Vulnerability and Exposures) e concentrarsi su CWE (Common Weakness Enumeration) per aiutare a comprendere meglio la prevalenza dei problemi.
Il report non si ferma qui, suggerisce anche di utilizzare i cosiddetti linguaggi di programmazione memory-safe per mitigare tali vulnerabilità, identificando questo approccio come il più efficiente, esplorando al contempo approcci complementari, tra cui l'utilizzo di hardware memory-safe e l'implementazione di metodi formali in alcuni casi specifici.
L'indicazione di utilizzare linguaggio di programmazione memory-safe di fatto mette in discussione l'utilizzo di uno dei linguaggi più famosi e più frequentemente impiegati dagli sviluppatori di tutto il mondo il C/C++. Questo linguaggio mette a disposizione la più ampia libertà di gestione della memoria e dei dispositivi hardware con massima efficienza sia in termini di prestazioni che di ottimizzazione dell'utilizzo delle risorse. Per questi motivi C/C++ rappresenta il "core" della maggior parte delle applicazioni.
Proprio i suoi punti di forza diventano i sui maggiori punti di debolezza quando analizziamo il codice valutandone gli aspetti di vulnerabilità. Il linguaggio C/C++ non si può considerare un linguaggio memory-safe in quanto proprio in nome della libertà di gestione delle risorse ne viene lasciato il pieno controllo allo sviluppatore il quale oltre a risolvere il problema dato, individuazione dell'algoritmo risolutivo, deve anche preoccuparsi di gestire operazioni di "contorno", per esempio allocare e deallocare la memoria, attività che in altri linguaggi non è necessaria perché gestita dal linguaggio stesso.

## Le più comuni vulnerabilità di sicurezza del C/C++

### *Buffer Overflow*

Un buffer overflow si verifica quando un programma scrive più dati in un buffer (una zona di memoria allocata) di quanto sia stato allocato per esso, sovrascrivendo così la memoria adiacente. Questo può portare a:

- Corruzione della memoria: sovrascrittura di dati importanti o indirizzi di ritorno, potenzialmente causando il crash del programma o un comportamento imprevedibile.
- Esecuzione di codice arbitrario: gli attaccanti possono sfruttare il buffer overflow per eseguire codice arbitrario inserito nella memoria durante l'overflow.

### *Uso di Puntatori Non Validi (Dangling Pointers)*

I puntatori non validi sono puntatori che fanno riferimento a una locazione di memoria che è stata deallocata o che non è più valida. L'utilizzo di tali puntatori può portare a:

- Crash del programma: tentare di accedere a memoria deallocata spesso provoca errori di segmentazione.
- Vulnerabilità di sicurezza: un attaccante potrebbe manipolare il programma in modo che punti a una zona di memoria controllata dall'attaccante stesso.

### *Memory Leak (Perdita di Memoria)*

Un memory leak si verifica quando un programma non libera memoria allocata dinamicamente che non è più in uso. I problemi principali includono:

- Esecuzione prolungata: nel tempo, il programma può esaurire la memoria disponibile, portando a un rallentamento delle

prestazioni o al crash.
- Problemi di disponibilità: in ambienti di server o applicazioni critiche, i memory leak possono ridurre la stabilità e la disponibilità del sistema.

*Use After Free*

L'uso dopo la deallocazione (use after free) si verifica quando un programma continua a utilizzare un puntatore dopo che la memoria alla quale puntava è stata liberata. Questo può portare a:

- Comportamento indefinito: accedere a memoria che è stata deallocata può portare a risultati imprevedibili o crash.
- Vulnerabilità di sicurezza: un attaccante potrebbe essere in grado di allocare nuovamente la memoria liberata e manipolare i dati in essa contenuti.

*Double Free (Doppia Deallocazione)*

Un doppio free si verifica quando un programma tenta di deallocare una zona di memoria che è stata già deallocata. Questo può causare:

- Crash del programma: il gestore della memoria potrebbe rilevare un'inconsistenza e terminare il programma.
- Possibili exploit di sicurezza: in alcuni casi, un attaccante può sfruttare la doppia deallocazione per manipolare la struttura interna del gestore della memoria e ottenere l'esecuzione di codice arbitrario.

*Race Condition*

Una race condition si verifica quando il comportamento del programma dipende dalla temporizzazione relativa di eventi concorrenti (come thread o processi multipli), e l'esito corretto non è garantito. Questo può portare a:

- Comportamento imprevedibile: variabili condivise potrebbero essere lette o scritte in modi non previsti.
- Vulnerabilità di sicurezza: un attaccante potrebbe sfruttare una race condition per modificare dati critici o ottenere un'escalation di privilegi.

*Integer Overflow e Underflow*

Gli overflow e underflow degli interi si verificano quando una variabile intera eccede il suo valore massimo rappresentabile o va sotto il suo valore minimo rappresentabile. Questo può causare:

- Comportamento imprevisto: operazioni aritmetiche possono dare risultati errati o comportamenti inaspettati.
- Vulnerabilità di sicurezza: un attaccante potrebbe sfruttare un overflow per bypassare controlli di sicurezza o causare altre vulnerabilità.

*Iniezione di Codice*

C++ consente l'uso di funzioni di sistema come system(), che possono essere vulnerabili a iniezioni di codice se non utilizzate correttamente. Ad esempio:

- Esecuzione di comandi malevoli: se l'input dell'utente è passato direttamente a una chiamata di sistema senza adeguata sanitizzazione, un attaccante potrebbe eseguire comandi arbitrari.

*Formato String Vulnerability*

Questa vulnerabilità si verifica quando l'input dell'utente viene utilizzato come stringa di formato in funzioni come printf() senza la dovuta verifica. Un esempio classico:

- Crash del programma o esecuzione di codice arbitrario: un attaccante può utilizzare specificatori di formato per leggere o scrivere in memoria, eseguire codice o causare un crash.

*Insufficient Input Validation*

L'insufficiente convalida dell'input è un problema generale che può portare a molteplici vulnerabilità, come SQL injection, command injection, o buffer overflow. Assicurarsi sempre che l'input venga validato correttamente per tipo, lunghezza e formato.

**Buone pratiche di programmazione**

Per mitigare questi problemi di sicurezza, è essenziale adottare buone pratiche di programmazione come:

- Sanitizzazione e convalida dell'input: Verifica rigorosa dell'input dell'utente.
- Uso di funzioni sicure: Funzioni che limitano l'accesso alla memoria e gestiscono in modo sicuro i buffer.
- Utilizzo di strumenti di analisi del codice: Strumenti come AddressSanitizer, Valgrind, e compilatori con protezione degli stack possono aiutare a rilevare vulnerabilità.
- Gestione corretta della memoria: Applicare tecniche come RAII (Resource Acquisition Is Initialization) per gestire automaticamente

l'allocazione e la deallocazione delle risorse.

Uscendo un attimo dallo schema si potrebbe valutare come buona pratica di programmazione l'utilizzo del linguaggio più adatto alla risoluzione del problema, che nella nostra fattispecie è la mitigazione dei problemi di sicurezza, dunque un linguaggio memory-safe quale per esempio Rust, Java, Python oppure C#. Tra questi Rust è ad oggi il linguaggio che presenta più similitudini rispetto a C/C++, con in più la prerogativa della gestione sicura della memoria.

## Storia del linguaggio Rust

Rust è un linguaggio di programmazione sviluppato da Mozilla Research, con l'obiettivo principale di fornire un'alternativa sicura e performante ai linguaggi tradizionali utilizzati per lo sviluppo a basso livello, come il C e il C++. Il progetto ha avuto inizio nel 2006, quando Graydon Hoare, un dipendente di Mozilla, iniziò a lavorare su un nuovo linguaggio di programmazione nel suo tempo libero, con l'intento di creare un linguaggio che potesse prevenire molti dei problemi di sicurezza e stabilità comuni nei linguaggi di programmazione tradizionali.
Nel 2009, Mozilla decise di sponsorizzare lo sviluppo di Rust, vedendo il potenziale per un linguaggio che potesse migliorare le prestazioni del loro browser Firefox, mantenendo al contempo un alto livello di sicurezza. Il linguaggio è stato progettato per essere "sicuro per default", ovvero per evitare comportamenti indefiniti e per garantire che molte categorie di bug di programmazione comuni, come le race condition e i puntatori nulli, fossero impossibili da rappresentare nei programmi scritti in Rust.
Nel 2010, il compilatore di Rust, inizialmente scritto in OCaml, fu riscritto in Rust stesso, una pratica comune conosciuta come "bootstrapping". Questa riscrittura permise di migliorare il compilatore e di ottimizzarlo utilizzando le caratteristiche uniche del linguaggio. Nel 2015, Rust diventa ufficialmente stabile e pronto per l'uso in produzione. Da allora, Rust è cresciuto rapidamente in popolarità, grazie alla sua enfasi sulla sicurezza della memoria e sulla concorrenza senza lock.

## Comparazione tra Rust e C++

### *Similitudini*

Da un punto di vista superficiale, Rust e C++ hanno una sintassi simile. Entrambi i linguaggi utilizzano un insieme affine di parole chiave e strutture di controllo, il che può rendere il passaggio da C++ a Rust agevole per molti sviluppatori. Entrambi i linguaggi offrono anche un controllo granulare sulla gestione della memoria, il che è fondamentale per i tipi di applicazioni a basso livello per cui sono stati progettati.
Entrambi i linguaggi supportano la programmazione orientata agli oggetti, permettendo agli sviluppatori di organizzare il codice in classi e oggetti per una migliore modularità e riutilizzo del codice. Infine, sia Rust che C++ hanno robusti sistemi di tipi che aiutano a prevenire una serie di errori di programmazione.

### *Differenze*

Nonostante queste similitudini, ci sono importanti differenze tra Rust e C++. Forse la più notevole è la sicurezza della memoria. C++ consente agli sviluppatori di gestire direttamente la memoria, libertà che può portare a errori gravi se non viene utilizzata correttamente. Rust, invece, adotta un approccio differente, combinando controllo statico al tempo di compilazione e un sistema di gestione della memoria sicuro che evita la necessità di un garbage collector utilizzando un sistema di ownership (proprietà) con borrowing (prestito) e lifetimes (durate) per garantire che ogni segmento di memoria abbia un proprietario unico, e che i riferimenti alla memoria siano sempre validi.

### *Allocazione della Memoria: Esempio di Confronto*

cpp — Copia codice

```cpp
#include <iostream>

int main() {
    int* arr = new int[10];  // Allocazione dinamica della memoria

    // Uso dell'array
    for (int i = 0; i < 10; i++) {
        arr[i] = i;
        std::cout << arr[i] << std::endl;
    }

    delete[] arr;  // Deallocazione manuale della memoria
    return 0;
}
```

In questo esempio C++, la memoria per l'array arr è allocata dinamicamente usando new, e deve essere esplicitamente deallocata usando delete[]. Se l'istruzione delete[] arr; fosse omessa o posizionata in modo errato, si verificherebbe un memory leak. Inoltre, un uso improprio di arr dopo la sua deallocazione porterebbe a un comportamento indefinito (use-after-free).

```rust
fn main() {
    let mut arr = vec![0; 10]; // Allocazione dinamica con gestione automatica della memor

    // Uso dell'array
    for i in 0..10 {
        arr[i] = i;
        println!("{}", arr[i]);
    } // 'arr' viene automaticamente deallocato qui
}
```

In Rust, l'allocazione dinamica della memoria per arr viene gestita dalla struttura Vec. Quando arr esce dallo scope (ambito), la memoria viene automaticamente deallocata. Questo meccanismo elimina completamente la necessità di gestire manualmente la memoria, prevenendo così molti dei bug di gestione della memoria comuni in C++.

*Gestione dei Tipi di Dato: Esempio di Confronto*

```cpp
#include <iostream>
#include <variant>

int main() {
    std::variant<int, double> var = 42;

    try {
        std::cout << std::get<int>(var) << std::endl;  // Accesso sicuro
        std::cout << std::get<double>(var) << std::endl;  // Eccezione std::bad_variant_ac
    } catch (const std::bad_variant_access& e) {
        std::cerr << "Errore: " << e.what() << std::endl;
    }

    return 0;
}
```

In questo esempio in C++, utilizziamo la classe std::variant per rappresentare un tipo di dato che può contenere un intero o un double. Se si tenta di accedere a un tipo diverso da quello attualmente contenuto, viene sollevata un'eccezione std::bad_variant_access. Questo tipo di gestione è relativamente sicuro, ma richiede comunque una gestione esplicita delle eccezioni per evitare il crash del programma.

```rust
fn main() {
    let var: Result<i32, &str> = Ok(42);

    match var {
        Ok(v) => println!("Valore: {}", v),
        Err(e) => println!("Errore: {}", e),
    }
}
```

In Rust, il tipo Result è utilizzato per rappresentare un valore che può essere di successo (Ok) o un errore (Err). L'uso di match per gestire i diversi possibili valori garantisce che ogni caso sia gestito esplicitamente, prevenendo errori di runtime e rendendo il codice più robusto e sicuro. Rust non ha eccezioni nel senso tradizionale, ma utilizza invece tipi di risultato come Option e Result per la gestione degli errori, incoraggiando un approccio di programmazione più sicuro e prevedibile.

## È questo il momento di abbandonare le applicazioni C/C++?

Dal mio punto di vista la domanda merita una ed una sola risposta, dipende!
Di certo avere a disposizione un linguaggio che eviti al programmatore di incappare nei tipici problemi di vulnerabilità che si possono incontrare lavorando in C++ non è cosa da sottovalutare, ma di certo non si può nemmeno sottovalutare il fatto che la base di codice già scritto in C++ è spesso enorme il che richiederebbe una smisurata quantità di tempo per essere convertito con tutti i problemi di possibili errori che una tale conversione comporterebbe.
Solo una attenta valutazione del progetto da realizzare e la conoscenza delle sue basi fondamentali (architettura, dipendenze, funzioni critiche) potrà fornirci la risposta che stiamo cercando. La sicurezza del codice non passa solo per l'utilizzo di un determinato linguaggio, ma si tratta piuttosto di analizzare i dati relativi alle vulnerabilità tipiche del linguaggio in uso e mettere in atto delle best practice di sviluppo sicuro per evitarle.
Un esempio per tutti, oltre il 30% delle vulnerabilità imputabili a errori nel codice C/C++ sono di classe CWE-119 ("buffer errors") mentre quelle di classe CWE-20 ("input validation") ammontano al 15%. Evitare i "buffer error" e validare correttamente l'input quando si sviluppa in linguaggio C/C++ significa sistemare buona parte dei rischi di vulnerabilità.
Un approccio allo sviluppo sicuro passa sicuramente per attente review e debug del codice, ma occorre ora più che mai giocare di anticipo affrontando i problemi sul nascere.

**www.rust-lang.org**

***Keywords: Rust; C++; sicurezza informatica; buffer overflow; Executive Order 14028; National Cybersecurity Strategy; CWE; CVE; memoria sicura; linguaggi memory-safe; Race condition; Memory leak; Dangling pointers; Use after free; Double free; Integer overflow; Iniezione di codice; string vulnerability; Mozilla; Garbage collector; Ownership; Borrowing***

# RUST: A SOLUTION TO C++ VULNERABILITIES?

The White House report from February 2023 encourages the adoption of secure programming languages to mitigate memory vulnerabilities, identifying Rust as an effective alternative to C/C++. Rust offers safe memory management through a system of ownership and compile-time checks, eliminating many common errors. However, C++ remains preferred for its performance and resource optimization, despite the associated security risks.
By Massimo Nannini

**Summary**



**The Report**

A report published this February by the White House shows how the U.S. government has urged software developers to abandon programming languages that cause buffer overflows and other vulnerabilities related to memory usage. The document explores the current reactive approach to cybersecurity and the potential burden on users. Essentially, it emphasizes the need for proactive cybersecurity approaches, focusing primarily on eliminating entire classes of vulnerabilities rather than merely fixing known ones.
The administration has introduced Executive Order 14028 along with the National Cybersecurity Strategy, which advocates rebalancing responsibilities and prioritizing long-term investments in cybersecurity, highlighting the importance of cooperation between government, industry, and the technical community in achieving these goals.
The document underscores the importance of adopting proactive measures to eliminate entire groups of existing software vulnerabilities. Specifically, it suggests that manufacturers should publish CVE (Common Vulnerabilities and Exposures) data and focus on CWE (Common Weakness Enumeration) to help better understand the prevalence of problems. The report does not stop there, also suggesting the use of so-called memory-safe programming languages to mitigate such vulnerabilities, identifying this approach as the most efficient, while also exploring complementary approaches, including the use of memory-safe hardware and the implementation of formal methods in specific cases.
The recommendation to use memory-safe programming languages effectively questions the use of one of the most famous and frequently used languages by developers worldwide, C/C++. This language offers the broadest freedom in memory and hardware device management with maximum efficiency in terms of both performance and resource optimization. For these reasons, C/C++ represents the "core" of most applications.
However, its strengths become its greatest weaknesses when analyzing the code for vulnerabilities. C/C++ cannot be considered a memory-safe language because, in the name of resource management freedom, full control is left to the developer, who, in addition to solving the given problem and finding the solution algorithm, must also manage "side" operations, such as allocating and deallocating memory—activities that are unnecessary in other languages because they are managed by the language itself.

**The Most Common C/C++ Security Vulnerabilities**

- ***Buffer Overflow***

A buffer overflow occurs when a program writes more data to a buffer (an allocated memory area) than it was allocated for, thereby overwriting adjacent memory. This can lead to:
o Memory corruption: Overwriting important data or return addresses, potentially causing program crashes or unpredictable behavior.
o Arbitrary code execution: Attackers can exploit the buffer overflow to execute arbitrary code inserted into memory during the overflow.

- ***Dangling Pointers***

Dangling pointers are pointers that refer to a memory location that has been deallocated or is no longer valid. Using such pointers can lead to:
o Program crashes: Attempting to access deallocated memory often causes segmentation faults.
o Security vulnerabilities: An attacker could manipulate the program to point to a memory

area controlled by the attacker.

- ***Memory Leak***

A memory leak occurs when a program does not free dynamically allocated memory that is no longer in use. The main issues include:

o Prolonged execution: Over time, the program may exhaust available memory, leading to performance slowdowns or crashes.

o Availability issues: In server environments or critical applications, memory leaks can reduce system stability and availability.

- ***Use After Free***

Use after free occurs when a program continues to use a pointer after the memory it pointed to has been freed. This can lead to:

o Undefined behavior: Accessing memory that has been deallocated can result in unpredictable outcomes or crashes.

o Security vulnerabilities: An attacker might be able to reallocate freed memory and manipulate the data it contains.

- **Double Free**

Double free occurs when a program attempts to deallocate a memory area that has already been deallocated. This can cause:

o Program crashes: The memory manager might detect an inconsistency and terminate the program.

o Potential security exploits: In some cases, an attacker can exploit double-free to manipulate the internal structure of the memory manager and execute arbitrary code.

- **Race Condition**

A race condition occurs when the program's behavior depends on the timing of concurrent events (such as multiple threads or processes), and the correct outcome is not guaranteed. This can lead to:

o Unpredictable behavior: Shared variables may be read or written in unintended ways.

o Security vulnerabilities: An attacker could exploit a race condition to modify critical data or gain privilege escalation.

- **Integer Overflow and Underflow**

Integer overflows and underflows occur when an integer variable exceeds its maximum representable value or goes below its minimum representable value. This can cause:

o Unexpected behavior: Arithmetic operations may produce incorrect results or unexpected behavior.

o Security vulnerabilities: An attacker could exploit an overflow to bypass security checks or cause other vulnerabilities.

- **Code Injection**

C++ allows the use of system functions such as system(), which can be vulnerable to code injection if not used correctly. For example:

o Execution of malicious commands: If user input is passed directly to a system call without proper sanitization, an attacker could execute arbitrary commands.

- **Format String Vulnerability**

This vulnerability occurs when user input is used as a format string in functions like printf() without proper verification. A classic example:

o Program crash or arbitrary code execution: An attacker can use format specifiers to read or write memory, execute code, or cause a crash.

- **Insufficient Input Validation**

Insufficient input validation is a general issue that can lead to multiple vulnerabilities, such as SQL injection, command injection, or buffer overflow. Always ensure that input is validated for type, length, and format.

**Best Programming Practices**

To mitigate these security issues, it is essential to adopt good programming practices such as:

- Sanitization and validation of input: Rigorous verification of user input.
- Use of secure functions: Functions that limit memory access and securely manage buffers.
- Use of code analysis tools: Tools like AddressSanitizer, Valgrind, and stack-protection compilers can help detect vulnerabilities.
- Proper memory management: Apply techniques like RAII (Resource Acquisition Is Initialization) to automatically manage resource allocation and deallocation.

Going beyond the usual practices, one could consider using the most suitable language for solving the problem, which in our case is mitigating security issues—thus, a memory-safe language such as Rust, Java, Python, or C#. Among these, Rust is currently the language that most closely resembles C/C++, with the added advantage of safe memory management.

## History of the Rust Language

Rust is a programming language developed by Mozilla Research, with the primary goal of providing a secure and performant alternative to traditional languages used for low-level development, such as C and C++. The project began in 2006 when Graydon Hoare, a Mozilla employee, started working on a new programming language in his spare time, aiming to create a language that could prevent many common security and stability issues found in traditional programming languages.

In 2009, Mozilla decided to sponsor the development of Rust, seeing its potential to improve the performance of their Firefox browser while maintaining a high level of security. The language was designed to be "safe by default," avoiding undefined behavior and ensuring that many categories of common programming bugs, such as race conditions and null pointers, were impossible to represent in programs written in Rust.

In 2010, the Rust compiler, initially written in OCaml, was rewritten in Rust itself, a common practice known as "bootstrapping." This rewriting improved the compiler and optimized it using the language's unique features. In 2015, Rust officially became stable and ready for production use. Since then, Rust has rapidly grown in popularity due to its emphasis on memory safety and lock-free concurrency.

## Comparison between Rust and C++

### *Similarities*

On the surface, Rust and C++ have a similar syntax. Both languages use a similar set of keywords and control structures, which can make transitioning from C++ to Rust easier for many developers. Additionally, both languages offer granular control over memory management, which is crucial for the types of low-level applications they were designed for. Both languages also support object-oriented programming, allowing developers to organize code into classes and objects for better modularity and code reuse. Finally, Rust and C++ both have robust type systems that help prevent various programming errors.

### *Differences*

Despite these similarities, there are significant differences between Rust and C++. Perhaps the most notable is memory safety. C++ allows developers to manage memory directly, which can lead to serious errors if not handled correctly. Rust, on the other hand, takes a different approach by combining static compile-time checks with a memory management system that avoids the need for a garbage collector. It uses an ownership system with borrowing and lifetimes to ensure that every memory segment has a unique owner and that memory references are always valid.

### *Memory Allocation: Example Comparison*

In this C++ example, memory for the array arr is dynamically allocated using new and must be explicitly deallocated using delete[]. If the delete[] arr; statement is omitted or incorrectly placed, a memory leak would occur. Additionally, improper use of arr after its deallocation would lead to undefined behavior (use-after-free).

In Rust, dynamic memory allocation for arr is handled by the Vec structure. When arr goes out of scope, the memory is automatically deallocated. This mechanism completely eliminates the need for manual memory management, thus preventing many of the common memory management bugs found in C++.

### *Data Type Management: Example Comparison*

In this C++ example, the std::variant class is used to represent a data type that can hold either an integer or a double. If an attempt is made to access a type other than the one currently contained, a std::bad_variant_access exception is raised. This type of management is relatively safe, but it still requires explicit exception handling to avoid a program crash.

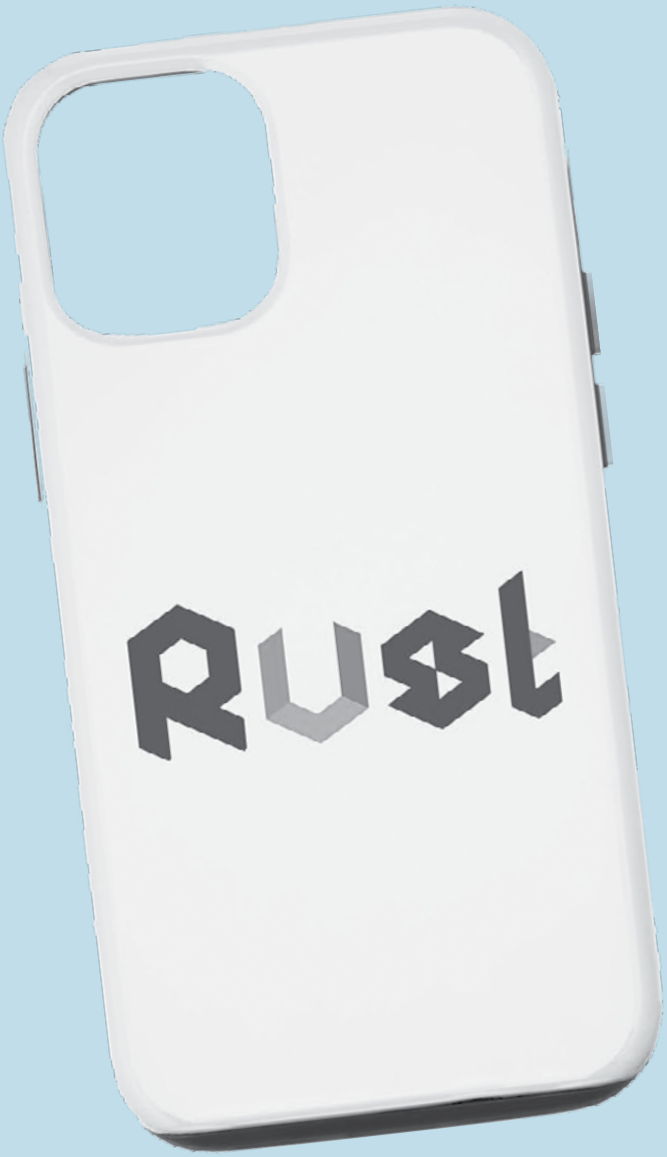
In Rust, the Result type is used to represent a value that can either be a success (Ok) or an error (Err). Using match to handle the different possible values ensures that each case is explicitly managed, preventing runtime errors and making the code more robust and secure. Rust does not have exceptions in the traditional sense but instead uses result types like Option and Result for error handling, encouraging a safer and more predictable programming approach.

### *Is It Time to Abandon C/C++ Applications?*

From my point of view, the question deserves only one answer: it depends! Having a language that prevents the programmer from encountering the typical security problems associated with C++ is certainly valuable. However, one cannot overlook the fact that the existing C++ codebase is often enormous, which would require an immense amount of

time to convert, with all the potential errors such a conversion might entail. Only a careful evaluation of the project to be developed and an understanding of its fundamentals (architecture, dependencies, critical functions) can provide the answer we seek. Code security does not only come from using a specific language; it is rather a matter of analyzing the vulnerabilities typical of the language in use and implementing secure development best practices to avoid them. For example, over 30% of vulnerabilities attributable to C/C++ code errors are of the CWE-119 class ("buffer errors"), while those of the CWE-20 class ("input validation") amount to 15%. Avoiding buffer errors and properly validating input when developing in C/C++ means mitigating a significant portion of the risk of vulnerabilities. An approach to secure development certainly involves thorough code reviews and debugging, but it is now more important than ever to be proactive by addressing problems at their source.

**www.rust-lang.org**




***Keywords: Rust; C++; cybersecurity; buffer overflow; Executive Order 14028; National Cybersecurity Strategy; CWE; CVE; memory safety; memory-safe languages; race condition; memory leak; dangling pointers; use after free; double free; integer overflow; code injection; string vulnerability; Mozilla; garbage collector; ownership; borrowing***

***(*) Massimo Nannini***
*IT engineer and business consultant,*
*info@gemaxconsulting.it*