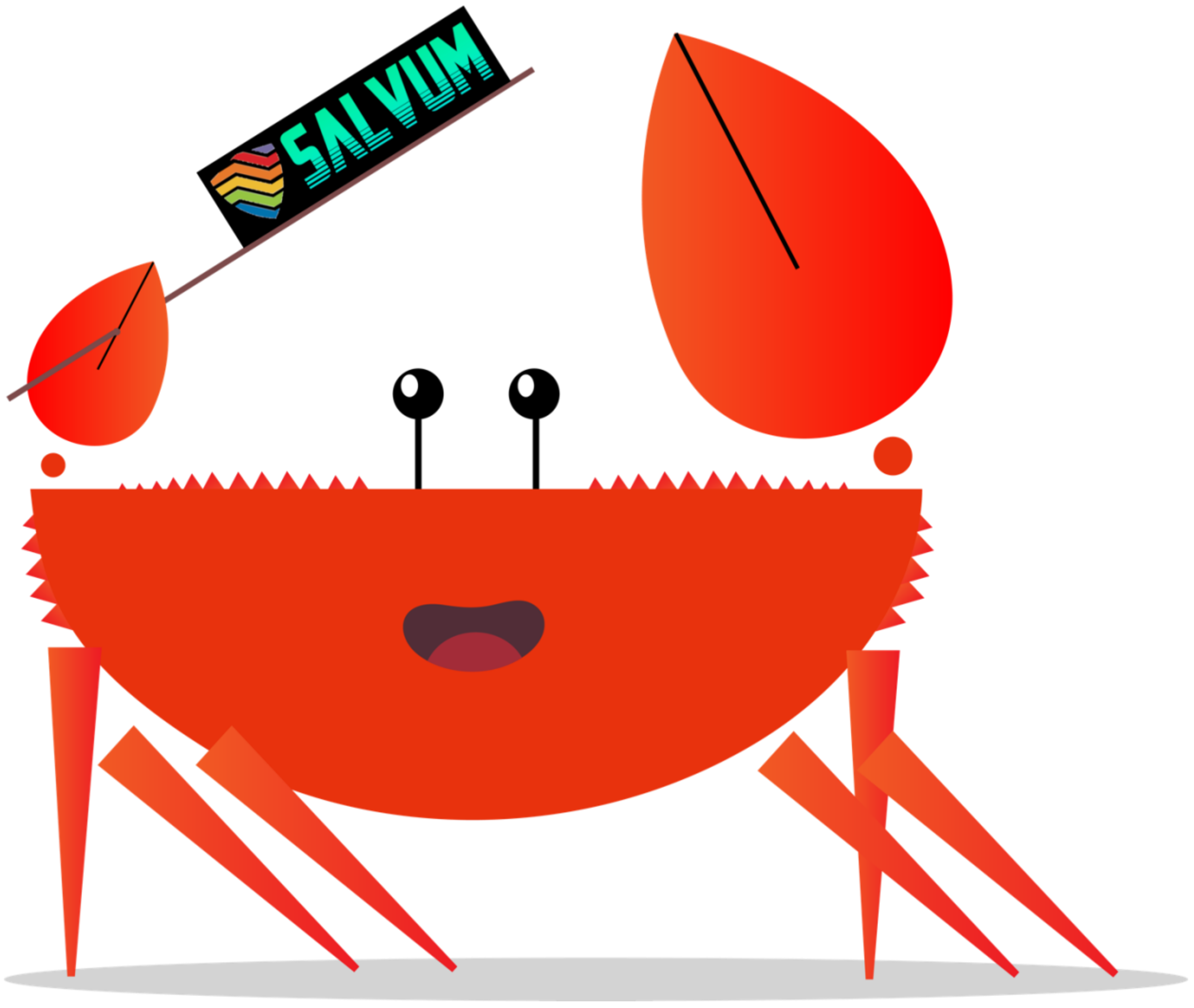
QVLx Labs

Protect and Empower

Why Salvum is so **Rusty**



Notes by $t@$h

Intro

In an era where mission safety and cybersecurity are more critical than ever, selecting a programming language for a project is a crucial decision. Our choice to write Salvum and many other software products in Rust was influenced by multiple factors including robust safety features and performance. This article explores Rust and what makes it an ideal choice for developing secure and reliable software.

Rust's inception, rooted in Graydon Hoare's vision in 2006, was driven by a desire to combine the performance and control of low-level languages like C and C++ with the safety and abstraction capabilities of high-level languages. Mozilla's sponsorship in 2009 was pivotal, leading to a collaborative community-driven development process.

Over the years, Rust evolved through rigorous community feedback and an open RFC process. This collaborative approach led to a language not only powerful and efficient but also adaptive to growing needs of modern software development. In my opinion, Rust's vibrant ecosystem is second to none and the biggest reason for its growth and adoption in the past decade.

Rustlang Design

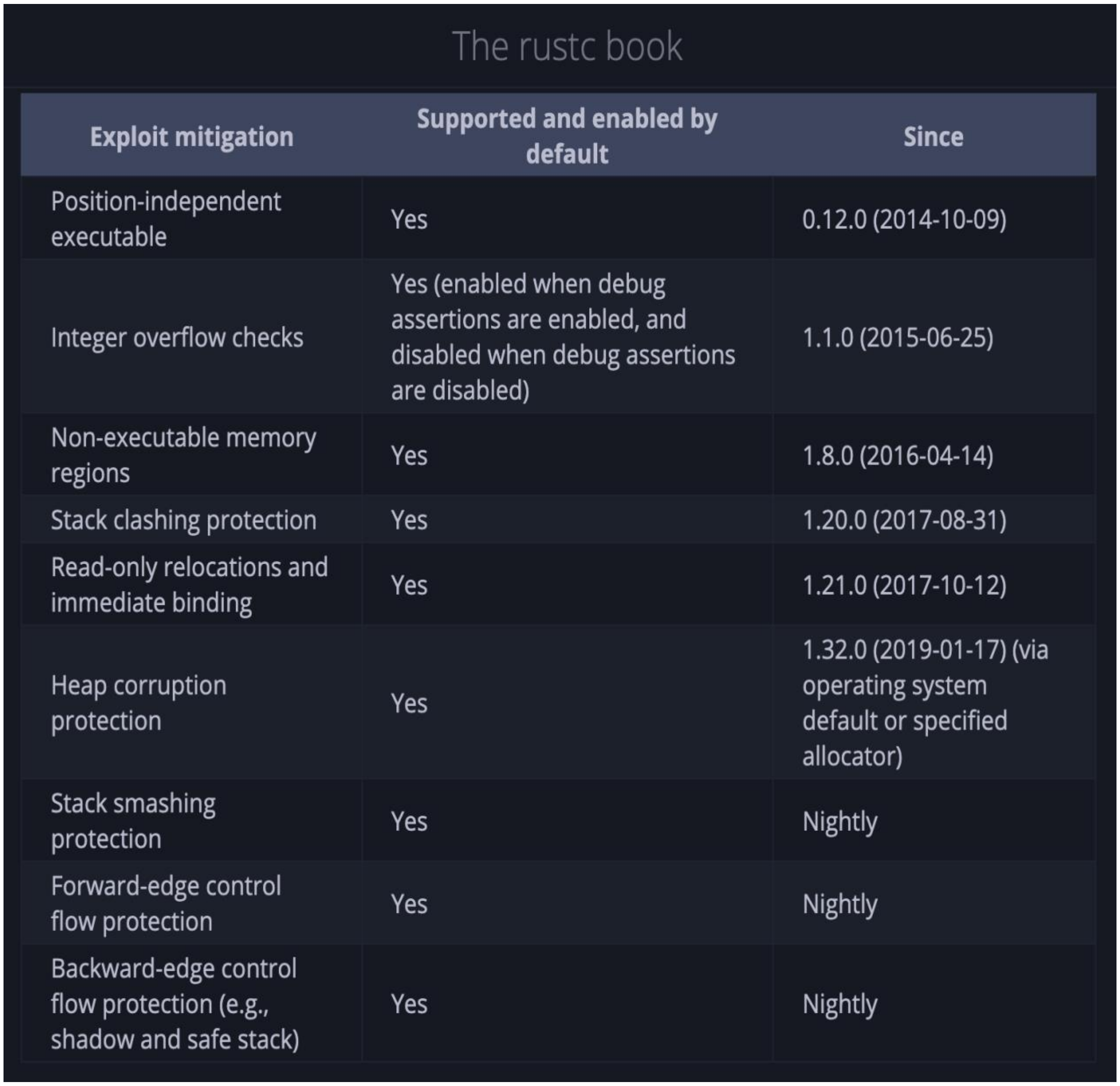
A cornerstone of Rust's design is its ability to ensure memory safety without the overhead of garbage collection common in many high-level languages. This is achieved through an elegant ownership model, where each datum has a single owner, and the compiler strictly enforces rules about data access.

Rust also addresses one of the most complex aspects of modern software development: safe concurrency. Its ownership model naturally extends to handling concurrent programming without fear, ensuring that data races, deadlocks, and other concurrency issues are caught at compile time. Rust even has support for asynchronous programming. An achievement only a handful of other languages even fewer system languages can claim.

The principle of zero-cost abstractions in Rust means that high-level constructs do not incur additional runtime overhead. The abstractions in Rust compile down to an efficient form like the hand-written code in lower-level languages like C. This makes Rust super performant, in fact comparable to other system languages like C and C++. In some cases even out-performing those long-time low-level languages.

Rust's ownership system is a unique solution to memory safety. Each variable in Rust has a single owner, and the scope of the variable defines its lifetime. The borrowing rules, which consist of references and mutable references, allow for safe, concurrent access to data, preventing race conditions and ensuring thread safety. Beautifully, all values are stack allocated by default. While this seems like it entails a lot of potential for stack smashing, nice synergy with LLVM makes it far less likely.

Rust’s approach to memory management at compile time is delicious. It enforces memory safety through its borrow checker, which validates references and ensures memory access patterns are safe, thus preventing common vulnerabilities like null pointer dereferences and buffer overflows. In fact here is a list of all of the security features that Rust has at the moment:



The rustc book

Exploit mitigation	Supported and enabled by default	Since
Position-independent executable	Yes	0.12.0 (2014-10-09)
Integer overflow checks	Yes (enabled when debug assertions are enabled, and disabled when debug assertions are disabled)	1.1.0 (2015-06-25)
Non-executable memory regions	Yes	1.8.0 (2016-04-14)
Stack clashing protection	Yes	1.20.0 (2017-08-31)
Read-only relocations and immediate binding	Yes	1.21.0 (2017-10-12)
Heap corruption protection	Yes	1.32.0 (2019-01-17) (via operating system default or specified allocator)
Stack smashing protection	Yes	Nightly
Forward-edge control flow protection	Yes	Nightly
Backward-edge control flow protection (e.g., shadow and safe stack)	Yes	Nightly

Rust's strong, static type system plays a crucial role in its safety guarantees. It helps catch errors early in development, reducing likelihood of runtime errors and vulnerabilities. The type system, with features like trait-based generics and pattern matching, offers flexibility and robustness in software design.

Error handling in Rust is designed to be explicit and robust. The Result and Option types in Rust force the programmer to handle the possibility of absence of value or the occurrence of error upfront, ensuring that such cases are addressed explicitly in the code. I have personally seen this language feature transform less-experienced programmers learn how to think about their code in terms of duality and thus empower them to always consider the case of code that behaves off nominally. This language design also builds confidence that the code will behave expectedly and QVLx associate researchers grew through this exposure.

Rust’s zero-cost abstractions mean that the abstractions provided by the language do not add any extra runtime cost. This is crucial in systems programming and other performance-critical applications where every cycle counts.

Rust’s compiler, again by leveraging LLVM’s backend, performs advanced optimizations that transform high-level constructs into efficient machine code. This capability allows developers to write high-level, abstract code without worrying about the performance implications often associated with high-level languages.

Cargo, Rust’s build system and package manager, streamlines the process of managing dependencies, compiling projects, and running tests. It ensures that the code uses the latest and most secure versions of libraries, a crucial aspect in maintaining the security of a software application. Such constructs help save so much time in development, which is a vital consideration to any endeavor that considers cost and hours. In addition, Rust also has built-in unit testing capability. And more advanced mocking can be achieved through third-party crates.

Clippy is a collection of lints for Rust, serving as an advanced, language-specific tool to catch common mistakes and improve code quality. It extends beyond the capabilities of traditional C/C++ linters by understanding Rust's unique features and idioms.

Rust offers a safer interface to system calls, preventing many security vulnerabilities. Its standard library provides abstractions that reduce risk of shell injection and other common pitfalls associated with calls in lower-level languages.

Rust’s approach to system interaction is built on its principles of memory and type safety. The language provides robust abstractions that encapsulate unsafe system interactions, ensuring that these interactions are both efficient and secure.

Rust allows developers fine-grained control over memory allocation and deallocation, crucial in low-level systems programming. While it automates memory management in safe code, it also provides the tools to manually manage memory when necessary, without sacrificing safety. Rust's approach to memory management is designed to prevent common errors found in other systems languages, such as memory leaks, double frees, and use-after-free vulnerabilities. Its compile-time checks ensure that memory is managed safely and efficiently.

Rust includes an unsafe keyword for instances where direct memory access is required, such as interfacing with other languages or performing certain low-level operations. This feature allows developers to perform these operations while explicitly marking these sections as unsafe, bringing attention to parts of the code that require careful review. Salvum core was written in pure Safe Rust. All wraps and expects were replaced with match statements that consider the case of error and don't panic but rather handle the error.

Rust's advanced type system, with features like traits and generics, contributes significantly to safety and security. They allow for expressive, reusable code without the runtime overhead typically associated with polymorphism and generics in other languages. Favor composition over inheritance. Seems Rustlang creators understood this principle well.

The decision to write a security engine in Rust was a no-brainer. And it has resulted in all of the many codebases developed by QVLx being able to be maintained by one person indefinitely.