Carnegie Mellon University Software Engineering Institute

Will Rust Solve Software Security?

JUNE 12, 2023

Joseph Sible, David Svoboda, Garret Wassermann

Document Markings

Agenda

- **Introduction**
- The Rust Security Model
- Limitations of the Rust Security Model
- Rust in the Current Vulnerability Ecosystem
- Rust Stability and Maturity
- Conclusion

Will Rust Solve Software Security?

Introduction

Carnegie Mellon University Software Engineering Institute

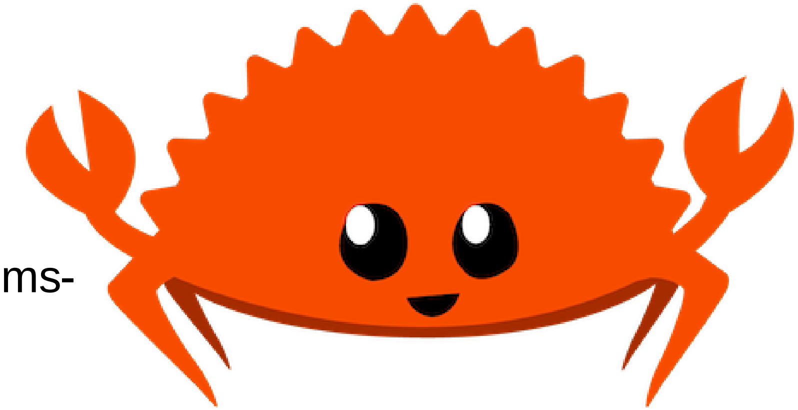
Background

- Easy to write vulnerable code in traditional languages
- Languages with strong type systems tended to be heavyweight and garbage collected
- Mozilla Research set out to create the best of both worlds

The Rust Language

A multi-paradigm systems-level language designed to eliminate certain kinds of security vulnerabilities.

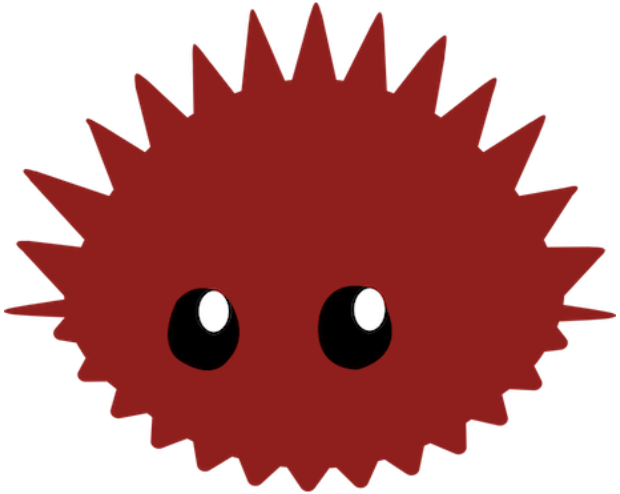
- Compiles directly to machine code, not interpreted
- No garbage collection making it attractive for systems-level programming
- Takes cues from modern languages
  - Functional paradigm supported (& often encouraged)
  - Robust typing system
  - Uses its own build system called cargo, with libraries/packages available at crates.io



Ferris the Crab: the unofficial Rust mascot (Released to public domain)

Safe Rust vs. Unsafe Rust

- Most code in Rust written entirely in Safe Rust
  - Exceptions: standard library implementation, C FFI, etc.
- Rust's safety guarantee:
  - Safe Rust can never cause Undefined Behavior
  - For Safe Rust, this is a guarantee
  - For Unsafe Rust, this is an obligation
  - Unsafe Rust must be *sound*



Corro the Unsafe Rusturchin
(Released to public domain)

The borrow checker

- Rust’s most significant contribution to programming is the borrow checker
- Every object is owned in one place, and can be borrowed for use elsewhere
- Borrows can be immutable (read-only) or mutable (read-write)
- The two key rules of borrowing:

1. Borrows must not outlast the original owner’s lifetime
2. Either a single mutable borrow or multiple immutable borrows may exist at any given time, but not both

Will Rust Solve Software Security?

The Rust Security Model

Carnegie Mellon University Software Engineering Institute

Comparison with other languages

- Traditional programming languages are often memory-unsafe
  - C
  - C++
- Memory safety used to require expensive runtime checks
  - Java
- Rust brings compile-time safety to the table

Iterator invalidation (C++11)

Carnegie Mellon University Software Engineering Institute

```
#include <cassert>
#include <iostream>
#include <vector>

int main() {
     std::vector<int> v{1,2,3};
     std::vector<int>::iterator it = v.begin();
     assert(*it++ == 1);
     v.push_back(4);
     assert(*it++ == 2);
}
```

Compiles without warnings.
Undefined Behavior at runtime!

Iterator invalidation (Rust)

```
fn main() {
    let mut v = vec![1, 2, 3];
    let mut it = v.iter();
    assert_eq!(*it.next().unwrap(), 1);
    v.push(4);
    assert_eq!(*it.next().unwrap(), 2);
}
```

Using an invalidated iterator

Iterator invalidation (Rust)


```
error[E0502]: cannot borrow `v` as mutable because it is also
borrowed as immutable
 --> rs.rs:5:5
  |
3 |         let mut it = v.iter();
  |                      -------- immutable borrow occurs here
4 |         assert_eq!(*it.next().unwrap(), 1);
5 |         v.push(4);
  |         ^^^^^^^^^ mutable borrow occurs here
6 |         assert_eq!(*it.next().unwrap(), 2);
  |                     ---------- immutable borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc --explain
E0502`.
```

Does not compile!
Runtime bug avoided

Use-after-free (C)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
    char *x = strdup("Hello");
    free(x);
    printf("%s\n", x);
}
```

Compiles without warnings.
Undefined Behavior at runtime!

Use-after-free (Rust)

Carnegie Mellon University Software Engineering Institute

```
fn main() {
      let x = String::from("Hello");
      drop(x);
      println!("{}", x);
}
```

x has already been freed here

Use-after-free (Rust)


```
error[E0382]: borrow of moved value: `x`
 --> src/main.rs:4:20
  |
2 |     let x = String::from("Hello");
  |         - move occurs because `x` has type `String`, which does not implement the `Copy` trait
3 |     drop(x);
  |          - value moved here
4 |     println!("{}", x);
  |                    ^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro `println` (in Nightly builds, run with -Z macro-backtrace for more info)

For more information about this error, try `rustc --explain E0382`.
```

Does not compile!
Runtime bug avoided

Other kinds of mistakes

Carnegie Mellon University Software Engineering Institute

- The billion dollar mistake: NULL
  - No such thing in Rust – uses Option instead. Compiler enforces:
    - Uses of optional variables must handle their absence
    - Non-optional variables must always have a value
- Concurrency
  - Data races
    - Prevented by having either exactly one writer, or an arbitrary number of readers
  - Mutexes, etc.
    - Present in other languages like C, but can be forgotten
    - Rust encodes their invariants into the type system, preventing misuse

Will Rust Solve Software Security?

Limitations of the Rust Security Model

Carnegie Mellon University Software Engineering Institute

Limitations

- Doesn't address features enabled by **`unsafe`** keyword (by design)
- Only addresses memory safety and concurrency safety (e.g. data races)
  - Doesn't address other security issues
    - SQL injections (and other injection attacks)
    - Floating-point errors
    - TOCTOU file race conditions (b/c not data race in memory)
    - Unsafe cryptography (e.g. MD5)
- Borrow Checker Limitations

Borrow Checker Limitation (C++11)

Carnegie Mellon University Software Engineering Institute

```
#include <cassert>
#include <iostream>
#include <vector>

int main() {
    std::vector<int> v{1,2,5};
    std::vector<int>::iterator it = v.begin();
    assert(*it++ == 1);
    v[2] = 3;
    assert(*it++ == 2);
}
```

Some operations invalidate C++ iterators but not member assignment

This code compiles, runs cleanly, and is memory-safe.

Borrow Checker Limitation (Rust)

Carnegie Mellon University Software Engineering Institute

Workarounds:

- Use the **vec<>::split_at_mut()** method
- Using indices rather than iterators
- Wrap vector elements in **std::cell**

```
fn main() {
    let mut v = vec![1, 2, 5];
    let mut it = v.iter();
    assert_eq!(*it.next().unwrap(), 1);
    v[2] = 3;
    assert_eq!(*it.next().unwrap(), 2);
}
```

immutable borrow of **v** occurs here

Memory-safe

mutable borrow of **v** occurs here

This code does not compile, it is rejected by the borrow checker

Rust Protection in Context

Protection	C	Java	Python	Rust
Memory Corruption	None	Full	Full	Full*
Integer Overflow	None	None	Full	Optional
Data Races	None	Some	None	Full*
Injection Attacks	None	Some	Some	Some
Misuse of 3rd-Party Code	None	None	None	None

*Full protection is offered for Rust code that does not use the `unsafe` keyword

Will Rust Solve Software Security?

Rust in the Current Vulnerability Ecosystem

Expanding Rust Use – and Attack Surface?

Carnegie Mellon University Software Engineering Institute

Rust is now being adopted by many important software projects.

- Rust support in the Linux kernel since 6.2 (January 2023)
- Rust support in Microsoft Windows code/API in May 2023
- Rust support for UEFI in progress
  - UEFI bytecode compilation target for rustc as of 2022
- Experimental GCC compiler for release in GCC 14 (early 2024)
  - The reference compiler **`rustc`** is built on LLVM

As Rust's popularity grows in industry, we can expect vulnerability and security issues around Rust to grow in importance.

CVEs and Vulnerability Analysis

A CVE search for "rust" in May 2023 yields over 400 entries, including vulnerabilities such as:

- Unsafe deserialization
- Integer overflow/underflow
- Out of bounds write, use after free, double drop (double free)
- Several forms of denial of service or memory leaks
- Various cryptographic issues – leaking secret data, incorrect use of cryptography keys/algorithms, etc.

While Rust's design helps prevent certain memory vulnerabilities, clearly many other kinds of vulnerabilities can still exist and require analysis.

Current Vulnerability Analysis Tools

Rust has a few experimental tools for code analysis:

- Rudra is an experimental static-analysis tool that can reason about certain classes of undefined behavior.
- Miri is an experimental Rust interpreter (dynamic analysis) that is designed to also detect certain classes of undefined behavior and memory access violations.

These tools are becoming standard analysis tools, though still experimental. Both have been used to discover CVEs and bugs in crates.io packages as well as in the rustc compiler itself.

However source code is not always available and requires specialized reverse engineering tooling.

Reverse Engineering Challenges

Carnegie Mellon University Software Engineering Institute

Rust's features ironically make more reliable/reproducible exploits.

There is evidence that malware authors are increasingly adopting Rust:

- In 2022, Rust code has been found in malware packages like BlackCat, Hive, RustyBuer, Luna

Source code is not always available to aid malware analysis, and there are gaps and challenges in providing Rust reverse engineering support

- Many tools assume C/C++ conventions and standards which are incorrect and/or not used by Rust
- Research needed in recognizing Rust code and abstractions in binary/machine code, and reconstructing those abstractions

Will Rust Solve Software Security?

Rust Stability and Maturity

Signs of Maturity

- ***Working reference implementation***
  - *Such as a compiler or interpreter*
- ***Complete written specification***
  - *Documents how the language is to be interpreted*
- ***Committee or group***
  - *Manages evolution of the language*
- ***Transparent process***
  - *for evolving the language*
- **Test suite**
  - Determines the compliance of third-party implementations
- **Meta-process**
  - Allows the committee to rate and improve its own processes
- **Technology**
  - Surveys how the language is being used in the wild
- **Repository**
  - of free third-party libraries

Table of Maturity

Language	C	Java	Python	Rust
First Appearance	1972	1995	1991	2010
Reference Implementation	None	JDK / HotSpot VM	cpython	rustc
Complete Specification	ISO/IES 9899:2017	JLS	Python Language Reference	The Rust Reference (incomplete)
Language Maintenance Group	ISO / IEC / JTC1 / SC22 / WG14	Sun , Oracle	Python Software Foundation	The Rust Project
Transparent Evolution Process	ISO	JCP	PEP Process	Request For Comments (RFC) process
Compliance Test Suite	Third-party commercial testsuites	JavaTest Harness	None	None
Meta-process to Improve Committee	ISO	None	None	None
Language Survey Technology	None	None	None	crater
Third-party Code Repository	None	None	Python Package Index (PyPI)	crates.io

Rust Stability Policies

- `crater` scans all code in crates.io and github.com
  - Any such code with a test that:
    - passes under the stable build
    - but fails under the nightly build
  - indicates a bug in the nightly build of the Rust compiler
    - or a change that would break code.
  - This is limited to OSS code on crates.io and github.com

- crates.io guarantees that crates will not become unavailable,
  - Even if they become deprecated. This prevents the left-pad fiasco.

- To use an experimental Rust feature, you must add:
  `#![feature(…)]`
  in your code.

Rust Maturity Conclusion

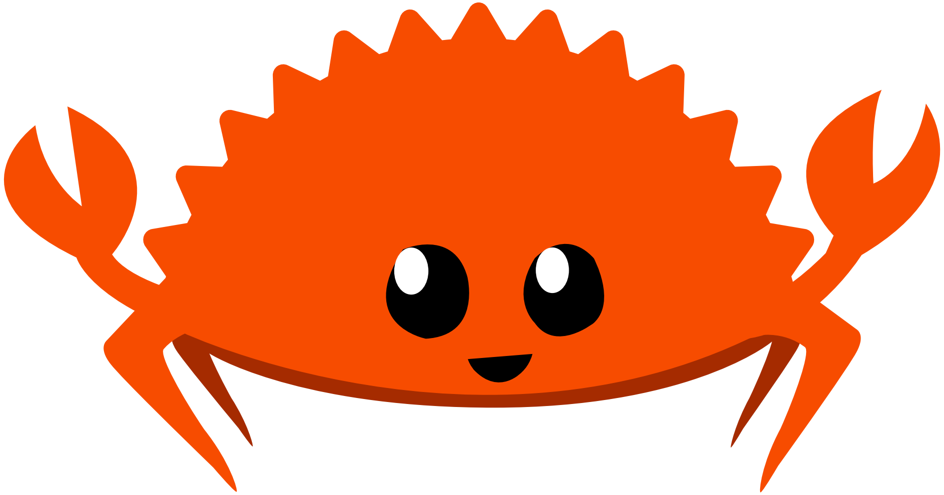
- For non-OSS Rust code:
  - Rust offers stability and maturity comparable to Python
    - The code might break when upgraded to a new version of Rust.
- **BUT** for OSS code published to crates.io and github.com
  - Rust's stability is considerably stronger
    - The code will not break on new versions of Rust without notification and the Rust community can provide assistance in fixing the code.
  - Rust's stability will be comparable to C or Java once Rust gains:
    - Complete written specification
      - GCC's proposed Rust extension should spur the Rust community to create a spec
    - Official compliance test suite

Will Rust Solve Software Security?

Conclusion

Carnegie Mellon University Software Engineering Institute

Key takeaways



- Rust is significantly safer than C, but it’s not a panacea
- Some vulnerabilities will probably never be totally preventable by a language
- Tooling is very good given how new Rust is, but it will still take time to be as rich as much older languages

Production readiness

- Being used in production by major companies such as Amazon and Google
- Lots of high-profile programs now use Rust in some capacity
  - Firefox
  - Linux kernel
  - Windows kernel

Contact Us

Joseph Sible
Associate Software Engineer

David Svoboda
Software Security Engineer

Garret Wasserman
Vulnerability Analyst

Email: info@sei.cmu.edu