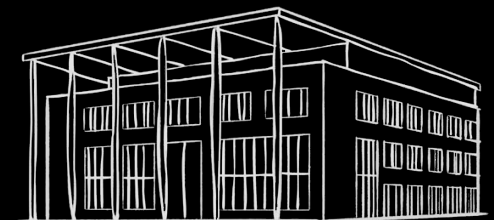




Learning Rust

Emphasizing Security and Performance

NEAL MAGEE
SCHOOL OF DATA SCIENCE
UNIVERSITY OF VIRGINIA



 **UVA Data Science**

Agenda

- Why Rust?
- What is Rust?
- Compare to other languages
 - OO vs. Procedural vs. Functional
- Basics
- Examples



Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection.



Software Memory Safety

Executive summary

Modern society relies heavily on software-based automation, implicitly trusting developers to write software that operates in the expected way and cannot be compromised for malicious purposes. While developers often perform rigorous testing to prepare the logic in software for surprising conditions, exploitable software vulnerabilities are still frequently based on memory issues. Examples include overflowing a memory buffer and leveraging issues with how software allocates and de-allocates memory. Microsoft® revealed at a conference in 2019 that from 2006 to 2018 70 percent of their vulnerabilities were due to memory safety issues. [1] Google® also found a similar percentage of memory safety vulnerabilities over several years in Chrome®. [2] Malicious cyber actors can exploit these vulnerabilities for remote code execution or other adverse effects, which can often compromise a device and be the first step in large-scale network intrusions.

Commonly used languages, such as C and C++, provide a lot of freedom and flexibility in memory management while relying heavily on the programmer to perform the needed checks on memory references. Simple mistakes can lead to exploitable memory-based vulnerabilities. Software analysis tools can detect many instances of memory management issues and operating environment options can also provide some protection, but inherent protections offered by memory safe software languages can prevent or mitigate most memory management issues. NSA recommends using a memory safe language when possible. While the use of added protections to non-memory safe languages and the use of memory safe languages do not provide absolute protection against exploitable memory issues, they do provide considerable protection.

What is Rust?

A memory-safe, compiled, statically typed, programming language that emphasizes performance and security.

Rust is a (mostly) *functional* language.

Functional vs. Procedural vs. OO

Rust is functional in that it focuses on *what* to compute; it prioritizes functions; its data is immutable; and it behaves more like an equation.

Concepts

Compiled vs. JIT Compiled

- Completely portable
- Bundled dependencies

Static Types

- Specified or Inferred
- CANNOT be changed

Garbage Cleanup

- Performs no GC like others
- Allows dev to manage memory
- No background processes

Memory Safe

- Ownership & Borrowing
- No Shared State
- Compile-time checks

Python

- Priority: Ease of Use
- Interpreted
- Dynamic typing
- Uses GC to reclaim memory

Rust

- Priority: Performance
- Compiled
- Static typing
- Ownership+Borrowing to limit memory access

Data Types

- Scalar types (single values)
 - Integers (whole, signed/unsigned)
 - Floating-point (fractional)
- Compound types (multiple values)
 - Tuples
 - Arrays
- Booleans
- Characters
- Strings
- Enums - define a type by enumerating its variants
- Structs - custom data types

Statically Typed Variables

Immutable

```
let my_name = "Neal";  
let word = String::from("blueberries");  
let x = 5;  
let z: i32 = 10;
```

Mutable

```
let mut my_name = "Neal";  
let mut x = 5;  
let mut pi: f64 = 3.1415927
```

Statically Typed Variables

Inferred

```
let m = 15;           // i32 default integer
let y = 3.14;         // f64 default flt-pt
let z = z.to_uppercase; // str inferred from usage
let bool2 = false;    // bool inferred
```

Explicitly Declared

```
let pi: f64 = 3.1415927
let x: i64 = 14;
let my_bool: bool = true;
let name = String::from("Phil");
```

Variables

```
fn main() {  
    // string using String::from()  
    let word = String::from("My favorite number is");  
    let num: i64 = 11;  
  
    // use the var in a print statement  
    println!("{}", word, num);  
}
```

Getting Started: cargo

Software

Start w/ Rustup

doc.rust-lang.org/book/ch01-01-installation.html

☆ 🔒 👤 ⌂

☰ ✎ 🔍

The Rust Programming Language

The Rust Programming Language

Foreword

Introduction

1. Getting Started

1.1. Installation

1.2. Hello, World!

1.3. Hello, Cargo!

2. Programming a Guessing Game

3. Common Programming Concepts

3.1. Variables and Mutability

3.2. Data Types

3.3. Functions

3.4. Comments

3.5. Control Flow

4. Understanding Ownership

4.1. What is Ownership?

4.2. References and Borrowing

4.3. The Slice Type

5. Using Structs to Structure Related Data

5.1. Defining and Instantiating Structs

5.2. An Example Program Using Structs

5.3. Method Syntax

6. Enums and Pattern Matching

6.1. Defining an Enum

6.2. The match Control Flow Construct

6.3. Concise Control Flow with if let

7. Managing Growing Projects with Packages, Crates, and Modules

7.1. Packages and Crates

7.2. Defining Modules to Control Scope and Privacy

7.3. Paths for Referring to an Item in the Module Tree

7.4. Bringing Paths Into Scope with the

Installation

The first step is to install Rust. We'll download Rust through `rustup`, a command line tool for managing Rust versions and associated tools. You'll need an internet connection for the download.

Note: If you prefer not to use `rustup` for some reason, please see the [Other Rust Installation Methods](#) page for more options.

The following steps install the latest stable version of the Rust compiler. Rust's stability guarantees ensure that all the examples in the book that compile will continue to compile with newer Rust versions. The output might differ slightly between versions because Rust often improves error messages and warnings. In other words, any newer, stable version of Rust you install using these steps should work as expected with the content of this book.

Command Line Notation

In this chapter and throughout the book, we'll show some commands used in the terminal. Lines that you should enter in a terminal all start with `$`. You don't need to type the `$` character; it's the command line prompt shown to indicate the start of each command. Lines that don't start with `$` typically show the output of the previous command. Additionally, PowerShell-specific examples will use `>` rather than `$`.

Installing rustup on Linux or macOS

If you're using Linux or macOS, open a terminal and enter the following command:

```
$ curl --proto 'https' --tlsv1.2 https://sh.rustup.rs -sSf | sh
```

The command downloads a script and starts the installation of the `rustup` tool, which installs the latest stable version of Rust. You might be prompted for your password. If the install is successful, the following line will appear:

<https://github.com/uvads/learn-rust>

Cargo basics

```
cargo new my-new-project
```

```
cargo run
```

```
cargo build / cargo build --release
```