AVAILABLE NOW

SHAHEER YASIR'S

RUSTSPLOIT

A PRACTICAL GUIDE FOR RUST PROGRAMMING LANGUAGE

RUSTSPLOIT

A PRACTICAL GUIDE FOR RUST PROGRAMMING LANGUAGE

SHAHEER YASIR

DISCLAIMER

This book, *RustSploit: Offensive Security with Rust – From Beginner to Advanced*, is intended for ethical hacking, research, and educational purposes only. Unauthorized use of the information, tools, or techniques in this book against any system, network, or device without explicit permission is illegal and punishable by law.

Legal Considerations

Readers must comply with cybersecurity laws, including:

- *Computer Fraud and Abuse Act (CFAA) (U.S.)* – Criminalizes unauthorized access, malware distribution, and credential trafficking. Violations can result in fines and imprisonment.
- *Computer Misuse Act (CMA) (UK)* – Outlaws unauthorized access, data interference, and system disruption. Penalties include imprisonment and fines.
- *Prevention of Electronic Crimes Act (PECA) (Pakistan)* – Criminalizes hacking, data theft, cyberterrorism, and system interference. Offenses may lead to heavy fines or imprisonment.

Ethical Use and Responsibility

This book does not promote cybercrime. Always:

- Conduct security testing only with proper authorization.
- Follow responsible disclosure policies.
- Comply with legal and ethical cybersecurity standards.

Author Disclaimer

The author, *Shaheer Yasir*, is not responsible for any misuse of the information, tools, or techniques presented in this book. The content is intended strictly for ethical hacking, penetration testing, and cybersecurity research conducted in legal and authorized environments. Any actions taken using the knowledge from this book are solely the responsibility of the reader. Unauthorized access, system compromise, or malicious use of the material may lead to legal consequences under national and international cybersecurity laws.Readers are expected to use this knowledge to enhance security, identify vulnerabilities, and improve defense mechanisms rather than for illegal or unethical activities. Ethical hackers and security professionals play a crucial role in protecting the digital world—act responsibly, follow legal guidelines, and always seek proper authorization before conducting any security testing.

ABOUT AUTHOR

Shaheer Yasir is an 18-year-old offensive security professional specializing in ethical hacking, penetration testing, and exploit development. He has been recognized by leading organizations such as **OpenAI, NASA, SpaceX, and Tesla** for his contributions to cybersecurity and vulnerability research. His passion for offensive security has driven him to master advanced hacking techniques through hands-on experience, self-learning, and mentorship from some of the industry's top experts.

Shaheer has gained knowledge under the guidance of **Hans Petrich**, a specialist in offensive security techniques, and **Rafay Baloch**, a globally recognized security researcher known for uncovering critical web vulnerabilities. Rafay made a name for himself in the cybersecurity world after discovering major browser security flaws affecting companies like Google and Microsoft, earning him international recognition.

In addition to his research and ethical hacking work, Shaheer is the founder of **SkynetSecurity**, a cybersecurity firm dedicated to helping businesses and individuals strengthen their digital security through penetration testing, vulnerability assessments, and advanced security solutions.

Shaheer has also authored two books:

- *The Secrets of Digital Underground* – A **comprehensive guide to ethical hacking**, covering everything from beginner to advanced techniques used in real-world cyber attacks.
- *The Art of Penetration Testing* – A **specialized book on web application security**, focusing on modern attack techniques, exploitation methods, and defensive countermeasures.

Beyond his technical contributions, Shaheer actively **engages with the cybersecurity community**, participating in **bug bounty programs, security conferences, and mentorship initiatives** to help others develop their skills in ethical hacking. His goal is to **bridge the gap between offensive and defensive security**, ensuring that organizations, researchers, and professionals stay ahead of emerging cyber threats. Through his work, he continues to **push the boundaries of digital security, inspire the next generation of ethical hackers, and contribute to a safer cyber world**.

Acknowledgements

Writing this book has been a journey of deep exploration into the world of **offensive security, ethical hacking, and cybersecurity research**. I am incredibly grateful to the **experts, professionals, and pioneers** who have contributed their knowledge, insights, and real-world experience to make this book a **valuable resource for cybersecurity enthusiasts and professionals alike**.

Special Thanks To:

Raed Ahsan – Pakistan’s Youngest Ethical Hacker & International Cybersecurity Speaker

Raed Ahsan is **Pakistan’s youngest ethical hacker**, a **cybersecurity researcher, Purple team specialist, and a globally recognized speaker**. He has spoken **three times at Black Hat Middle East**, one of the most prestigious cybersecurity conferences in the world, and has also been a featured speaker at **ISACA and other international cybersecurity forums**.

With a strong background in **penetration testing, exploit development, and advanced red teaming**, Raed has contributed to **securing enterprises, training professionals, and spreading cybersecurity awareness**. His expertise in **offensive security, vulnerability research, and cyber defense strategies** has made a significant impact in the industry. His insights into **modern attack vectors, ethical hacking methodologies, and practical cybersecurity applications** have greatly enriched the depth of this book.

Salman Aslam – Former FIA Cyber Crime Investigator & Digital Forensics Expert

Salman Aslam is a **former investigator at the Federal Investigation Agency (FIA) Cyber Crime Wing**, specializing in **digital forensics, cybercrime investigations, and threat intelligence**. With years of experience in **analyzing cyber threats, tracking cybercriminals, and handling complex digital investigations**, he has played a key role in **law enforcement’s fight against cybercrime**.

His deep expertise in **malware analysis, forensic investigations, and cybersecurity law enforcement** has provided this book with **practical, real-world case studies and methodologies** used by professionals in **cybercrime investigations**. His knowledge has helped bridge the gap between **offensive security and legal frameworks**, ensuring that ethical hacking is understood from both **a technical and a legal perspective**.

Final Words of Appreciation

This book is the result of **countless hours of research, collaboration, and hands-on testing**. The field of **cybersecurity and ethical hacking** is ever-evolving, and it is through the dedication of **industry leaders, researchers, and practitioners like Raed Ahsan and Salman Aslam** that we continue to push the boundaries of **offensive security and digital defense**.

To **all the readers, learners, and future cybersecurity experts**—this book is for you. Keep exploring, keep innovating, and remember: **cybersecurity is not just a skill, it's a responsibility.**

🚀 **Welcome to the world of hacking and offensive security!** 🔥

A MESSAGE FOR BEGINNERS

Dear Reader,

If you're new to programming or have struggled with it before, let me tell you something: **you are not alone.** Many of the best hackers and security researchers started from zero. Maybe you've tried coding before and thought, *this is too hard* or *I just don't get it.* Maybe you even hate programming. But here's the truth:

Programming is not about talent—it's about persistence.

The problem isn't that you "can't code." The problem is that you haven't had the right mindset. **Programming is just like learning a new language**—at first, it's confusing, but once you understand the syntax and logic behind it, you are **good to go**. You don't need to memorize everything. You just need to understand how things work, and from there, you can build anything.

I know this struggle because I've been there myself. There were times I struggled with even the **most basic installations,** times when things just wouldn't work no matter what I tried. I spent hours debugging, feeling frustrated, but I **never chose to quit.** I realized that in cybersecurity, **you are on your own.** No one is going to hold your hand. No one will spoon-feed you the answers. **It's just you vs. your laziness.**

So if you're thinking, *I don't know if I can do this*, **shut that voice down right now.** You absolutely **can do this.** The difference between those who succeed and those who fail is **who refuses to quit.** You have the potential, but you need to **get up and start working.**

Think of it like going to the gym. You don't walk in on Day 1 and lift the heaviest weights. You start light, build strength, and before you know it, you're doing things you once thought were impossible. **The same applies to hacking and programming.** David Goggins talks about **pushing past the limits your mind sets on you.** Your brain will tell you that you're not smart enough, not talented enough, not "meant" for this. **That's all bullshit. You are more capable than you realize.**

So right now, I want you to make a decision: **Are you going to be the person who gives up at the first challenge, or the one who keeps pushing forward until you master it?**

This book is designed to take you from **zero to advanced** in Rust-based offensive security. Follow along, take notes, write code, break things, and learn from your mistakes. **Don't rush, don't compare yourself to others—just focus on improving every single day. No one is coming to save you. You are on your own in this field. Get up and start working.**

I believe in you. Now, let's get to work.

Shaheer Yasir,

BOOK OUTLINE : RUSTSPLOIT

Preface

- **Why Rust? The need for memory-safe, low-detection, high-performance hacking tools.**
- **Rust's advantages in offensive security (memory safety, stealth, cross-platform, async networking).**
- **Who is this book for? Beginners to advanced security professionals.**
- **Structure of this book: Theory, hands-on labs, and real-world attack scenarios.**
- **Setting up a Rust offensive security lab: Installing Rust, using Linux VMs, configuring testing environments.**

Chapter 1: Introduction to Rust for Offensive Security

- **Why Rust is safer, faster, and more powerful than C/C++ for hacking.**
- **Comparing Rust to Python, Go, and C in offensive security.**
- **Installing Rust (rustup, cargo, rustc) on Linux, Windows, and macOS.**
- **Setting up a hacker-friendly Rust development environment.**
- **First hacking tool: A simple Rust-based port scanner.**

Chapter 2: Rust Fundamentals for Cybersecurity

- **Rust syntax basics: variables, functions, structs, traits.**
- **Understanding Rust's Ownership and Borrowing Model (prevents memory corruption).**
- **Error handling in Rust: Result, Option, unwrap.**
- **Using unsafe Rust for security research.**
- **Project: Writing a basic network reconnaissance tool.**

Chapter 3: Advanced Rust for Hackers

- **Multithreading and async Rust (tokio, async-std) for high-performance tools.**
- **Interfacing with C libraries using bindgen for low-level hacking.**
- **Writing efficient file I/O tools for malware development.**
- **Project: Writing a simple Rust-based HTTP reconnaissance tool.**

Chapter 4: Reconnaissance and OSINT with Rust

- **Building a stealthy subdomain enumeration tool.**

- **Automating WHOIS lookups and metadata extraction.**
- **Parsing and analyzing DNS records with Rust.**
- **Project: Developing a Rust-based passive reconnaissance tool.**

Chapter 5: Writing Rust-Based Exploits

- **Understanding memory corruption vulnerabilities in Rust.**
- **Using unsafe Rust to bypass protections.**
- **Writing buffer overflow exploits in Rust.**
- **ROP (Return-Oriented Programming) techniques with Rust.**
- **Project: Developing a simple PoC exploit in Rust.**

Chapter 6: Networking Attacks and Covert Channels

- **Creating a stealthy Rust TCP/UDP backdoor.**
- **Implementing DNS tunneling for covert communication.**
- **Crafting raw network packets using pnet.**
- **Performing MITM (Man-in-the-Middle) attacks using Rust.**
- **Project: Writing a Rust-based HTTP proxy for intercepting and modifying traffic.**

Chapter 7: Payload Development and Shellcode Execution

- **How Rust interacts with Windows APIs (winapi crate).**
- **Executing shellcode using VirtualAlloc and CreateRemoteThread.**
- **Implementing process injection techniques (DLL injection, process hollowing).**
- **Project: Developing a Rust-based reflective DLL loader.**

Chapter 8: Rootkits and Persistence Techniques

- **Hooking system calls in Linux and Windows.**
- **Writing a simple Linux rootkit using Rust and eBPF.**
- **Hiding Rust malware from process monitoring tools.**
- **Creating persistence via Windows registry and scheduled tasks.**
- **Project: Writing a stealthy Rust-based persistence mechanism.**

Chapter 9: Fileless Malware and Memory Injection

- **Executing payloads directly from memory (mmap, VirtualAlloc).**
- **Reflective PE injection with Rust.**
- **Writing self-modifying Rust malware for AV evasion.**

- **Project: Developing a fileless Rust loader that runs shellcode in-memory.**

Chapter 10: Ransomware Development in Rust

- **Understanding how ransomware works.**
- **Implementing strong encryption (AES, RSA) for data locking.**
- **Developing a Rust-based file encryptor and decryptor.**
- **Bypassing detection with Rust's low signature footprint.**
- **Project: Creating a controlled, ethical Rust ransomware simulation for research purposes.**

Chapter 11: Automating Privilege Escalation with Rust

- **Identifying privilege escalation opportunities.**
- **Developing UAC bypass techniques in Rust.**
- **Token impersonation and Windows access control manipulation.**
- **Project: Writing an automated Rust-based privilege escalation checker.**

Chapter 12: Web Exploitation with Rust

- **Writing an automated SQL injection scanner.**
- **Developing a Rust-based SSRF (Server-Side Request Forgery) scanner.**
- **Building a Burp Suite-like HTTP proxy in Rust.**
- **Project: Creating an XSS and CSRF detection tool in Rust.**

Chapter 13: Reverse Engineering and Fuzzing with Rust

- **Parsing PE and ELF binaries using Rust (goblin crate).**
- **Developing a custom Rust-based disassembler.**
- **Automating crash analysis and exploit development with Rust.**
- **Project: Writing a Rust-based binary fuzzer to find vulnerabilities.**

Chapter 14: Bypassing AV, EDR, and Sandboxes

- **Understanding how modern AV/EDR detects malware.**
- **Process injection techniques that bypass behavioral analysis.**
- **Rust-based polymorphic malware and self-modifying code.**
- **Detecting and bypassing sandboxes with Rust system fingerprinting.**
- **Project: Writing a Rust-based AV evasion tool.**

Chapter 15: Building a Stealthy Rust-Based C2 Framework

- **Designing a Command and Control (C2) framework in Rust.**
- **Implementing encrypted communication (TLS/WebSockets).**
- **Deploying Rust agents for stealthy remote execution.**
- **Building modular exploit delivery mechanisms.**
- **Project: Writing a minimal Rust-based C2 with remote shell functionality.**

Chapter 16: Covert Data Exfiltration and Stealthy Communications

- **Exfiltrating data using ICMP, DNS, and HTTP tunnels.**
- **Implementing end-to-end encryption for stealthy exfiltration.**
- **Detecting and bypassing network monitoring solutions.**
- **Project: Writing a Rust-based covert exfiltration tool.**

Final Notes: Ethical Considerations and Responsible Disclosure

- **The ethics of offensive security and red teaming.**
- **Why responsible disclosure matters.**
- **How Rust can be used for both offense and defense.**
- **Encouraging contributions to the open-source Rust security community.**

PREFACE

Why Rust?

Rust is quickly becoming a **game-changer in cybersecurity**, especially in offensive security. Traditional programming languages like **C and C++** have been widely used for hacking tools and exploit development, but they come with **serious security risks** like buffer overflows and memory corruption vulnerabilities. Rust eliminates these risks while still offering **low-level system control, high performance, and powerful security features**.

Security researchers and red teamers are now shifting towards Rust because of its:

- **Memory safety** – Prevents vulnerabilities like buffer overflows and use-after-free errors.
- **Low detection rates** – Rust binaries are less common in malware analysis databases, making them harder to detect.
- **Cross-platform support** – Easily compiles for **Windows, Linux, and macOS**, making it ideal for writing payloads and exploits.
- **Concurrency and async networking** – Perfect for building fast, stealthy command-and-control (C2) servers and exploit automation tools.

This book will guide you from the **basics of Rust** to **advanced offensive security techniques**, equipping you with **real-world hacking skills**.

Rust's Advantages in Offensive Security

Rust is not just another programming language—it's a **powerful tool for hackers**. Here's why:

1. **Memory Safety Without Performance Loss**
    - Unlike C/C++, Rust ensures **safe memory management without needing garbage collection**.
    - This means you can write **efficient, high-speed exploits and payloads** without memory corruption risks.
2. **Stealth and Low Detection**
    - Most modern security tools and antivirus programs focus on **detecting C, C++, and Python-based malware**.
    - Rust-based malware is **rarely analyzed**, making it a great choice for stealthy offensive operations.
3. **Cross-Platform Capabilities**
    - Rust can **compile for Windows, Linux, macOS**, and even embedded systems.
    - This means you can write **payloads, rootkits, and hacking tools** that work across multiple targets.

4. **Powerful Async Networking**
    - With **Tokio** and **async-std**, Rust can handle **high-speed network scanning, C2 communications, and covert data exfiltration** without performance issues.

Who Is This Book For?

This book is designed for a wide range of readers:

- **Beginners** – Those with little or no Rust experience but want to learn offensive security.
- **Experienced Hackers** – Those familiar with cybersecurity who want to harness Rust's power.
- **Developers & Red Teamers** – Those looking for **stealthier, high-performance hacking tools** that bypass modern defenses.

No matter your experience level, this book will take you from **basic Rust programming to advanced hacking techniques** used in real-world offensive security.

How This Book Is Structured

This book is divided into multiple sections that progressively build your **understanding of Rust and its applications in hacking**.

1. **Introduction to Rust** – Covers the basics of Rust, including syntax, memory management, and system programming.
2. **Offensive Security with Rust** – Explores **networking attacks, exploit development, and reconnaissance**.
3. **Malware Development** – Focuses on **rootkits, payload execution, persistence, and stealth**.
4. **Red Team Operations** – Covers **privilege escalation, web exploitation, and command-and-control (C2) frameworks**.
5. **Advanced Topics** – Includes **bypassing AV/EDR, ransomware simulation, and covert data exfiltration**.

Each chapter includes **hands-on projects** to help you apply what you've learned in **real-world hacking scenarios**.

Setting Up Your Rust Offensive Security Lab

Before diving into the book, you need a proper **testing environment**. Offensive security requires a **safe and legal space to test exploits and malware** without breaking any laws.

1. Install Rust

Rust is easy to install on any operating system. Use the following command to install Rust and its package manager (`cargo`):

Linux & macOS:

bash
Copy
```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source $HOME/.cargo/env
```

Windows:

1. Download and install **Rustup** from rust-lang.org.

Open PowerShell and run:
powershell
Copy
```
rustup update
```

2.

Verify installation:

bash
Copy
```
rustc --version
cargo --version
```

2. Set Up a Virtualized Lab

To safely test hacking tools, set up a **virtual lab** using:

- **VirtualBox** or **VMware** – To create isolated virtual machines.
- **Kali Linux or Parrot OS** – Preloaded with hacking tools.
- **Windows 10 VM** – Useful for testing Windows-based exploits.

Ensure **networking is configured properly** for attack simulations (NAT, Bridged, or Host-only).

---

3. Install Essential Hacking Tools

Once your lab is set up, install these tools:

- **Wireshark** – For packet analysis.
- **nmap & RustScan** – For network scanning.
- **Metasploit** – For exploit testing.
- **Burp Suite** – For web application security testing.

With your **Rust environment and hacking lab ready**, you are now set to begin your journey into **offensive security with Rust**. Let’s get started

CHAPTER 1 : Introduction to Rust for Offensive Security

Figure 1.1 (image sourced from google)

Why Rust is Safer,Faster,Better than C++ programming language

If you're interested in hacking, cybersecurity, or writing low-level code, you've probably heard of C and C++. These two programming languages have been around for decades, powering everything from operating systems to hacking tools. But now, a new language is changing the game: Rust. And it’s not just hype—Rust is actually **safer, faster, and more powerful** than C and C++ in many ways. Let’s break it down in **super simple terms**.

C and C++ are powerful, but they’re also dangerous. They give you direct control over the computer’s memory, but that also means you can **easily make mistakes**. These mistakes can lead to crashes, security holes, and serious vulnerabilities that hackers love to exploit.

For example, in C or C++, you might accidentally **use memory after it has already been freed**, or **write data into places it doesn't belong**. This can cause things like **buffer overflows** or **use-after-free** vulnerabilities, which attackers can use to take over a system.

Rust completely **eliminates** these risks. It has a built-in safety system that **stops you from making these mistakes** in the first place. The language won't even let your program run if it detects a potential memory problem. That means fewer crashes, fewer security holes, and way less debugging.

Imagine C/C++ as a **sharp knife with no handle**—you can cut things quickly, but you might also cut yourself. Rust is like **a smart knife that only lets you cut what you're supposed to, without the risk of injury**.

Rust Helps You Write Better Hacking Tools

Hackers and cybersecurity experts use C and C++ to build tools like **network scanners, password crackers, exploits, and malware**. Rust lets you build the exact same things, but with **fewer bugs and better reliability**.

With Rust, you can write tools that:

- Run fast and efficiently, just like in C or C++.
- Are much harder to crash because Rust **prevents dangerous memory errors**.
- Are harder for security teams to reverse-engineer, making them more stealthy.

If you were writing a hacking tool in C++, you'd have to manually manage memory, watch out for buffer overflows, and carefully avoid security holes. In Rust, the **language takes care of all of that for you**, so you can focus on building powerful tools without worrying about hidden bugs.

Imagine C/C++ as **writing a secret message on a piece of paper**—if someone finds it, they can read it easily. Rust is like **writing a secret message in invisible ink**—it's much harder for people to analyze and understand.

Rust Prevents Common Hacking Bugs

Many of the biggest vulnerabilities in hacking come from **programming mistakes in C and C++**. Rust **automatically prevents** a lot of these mistakes, making it much harder for hackers to exploit programs written in Rust.

For example, C and C++ often have **buffer overflows**, which happen when a program writes too much data into a memory space, overwriting other important information. Hackers **love buffer overflows** because they can use them to inject their own code and take control of a system. Rust **stops buffer overflows from happening** by design.

Another common issue in C and C++ is **use-after-free**, where a program tries to use memory that's already been deleted. This often leads to crashes or security vulnerabilities. Rust

completely eliminates use-after-free errors, making it much harder to exploit programs written in Rust.

Think of C/C++ as **a house with no locks**—anyone can just walk in. Rust is **a house with an automatic security system that stops intruders before they even reach the door**.

Rust is Harder to Reverse Engineer (Good for Hackers & Red Teams)

If you write a hacking tool in C or C++, security researchers can often **reverse-engineer** it, meaning they can analyze how it works and find ways to detect or block it. Rust makes this much harder.

Rust compiles code in a way that **confuses reverse-engineering tools**, making it difficult for security teams to figure out what your program is doing. This makes Rust **a great choice for red team tools, exploits, and stealthy hacking programs**.

Imagine C/C++ as **a locked diary**—someone might still be able to break in and read it. Rust is like **a self-destructing secret message**—once it's sent, it's almost impossible to recover.

Rust is Growing Fast in Cybersecurity

More and more **security professionals and hackers** are switching to Rust because it's safer, more reliable, and just as powerful as C and C++. Some well-known security tools already use Rust, like:

- **RustScan** – A super-fast port scanner, like Nmap but quicker.
- **FeroxBuster** – A web directory scanner for finding hidden files and folders.
- **Nuqleon** – A Rust-based exploitation framework for red teams.

As Rust keeps growing, more hackers and security researchers will **move away from C and C++** and start using Rust for **exploits, malware, and security tools**.

Should You Learn Rust for Hacking?

If you want to write **fast, powerful, and stealthy hacking tools**, Rust is one of the best languages to learn. It gives you all the power of C and C++ but with **fewer bugs, better security, and more reliability**.

If you're looking for **a language that's easy to learn** and you just want to write quick scripts, Python is still a great choice. But if you want to build **serious hacking tools, malware, or low-level exploits**, Rust is the **future of cybersecurity programming**.

Rust isn't as popular as C and C++ yet, but **it's growing fast**. If you start learning Rust now, you'll be ahead of most hackers who are still using old-school C++.

Rust vs. Python, Go, and C in Offensive Security – Which Language is Best for Hacking?

Rust is like **a powerful, modern weapon**—it gives you all the **speed and control of C**, but without the **bugs and security risks**. It's becoming more popular in cybersecurity because it allows hackers to build **stealthy, high-performance tools** while avoiding common vulnerabilities.

Why Rust is Good for Hacking:

- **Memory Safety:** Rust **prevents** dangerous bugs like **buffer overflows and use-after-free errors**, which are often exploited in C programs.
- **Fast Execution:** Rust runs **as fast as C**, meaning you can build **high-performance malware, rootkits, and exploits**.
- **Harder to Reverse Engineer:** Unlike Python, Rust compiles into **machine code**, making it **harder to detect and analyze**.
- **Concurrency Support:** Rust makes it easy to write **multi-threaded tools**, which is useful for **brute-force attacks and network scanners**.

What Rust is Best For:

- Writing **advanced malware** and **stealthy exploits**.
- Creating **red team tools** that are **hard to detect**.
- Developing **high-performance brute-force or scanning tools**.

Downsides of Rust:

- **Harder to learn than Python**—Rust has strict rules that make programming safer, but also more complex.
- **Not as many libraries for hacking**—Unlike Python, Rust doesn't have built-in tools for networking and cryptography, so you have to write more code yourself.

Python – The Hacker's Favorite Scripting Language

Python is like a **Swiss Army knife for hackers—**it's **easy to learn**, has **tons of hacking libraries**, and is great for **automation and penetration testing**. It's the most commonly used language in cybersecurity because you can quickly write scripts for **network attacks, exploit development, and data analysis**.

Why Python is Good for Hacking:

- **Beginner-Friendly:** Python has **simple syntax**, making it **the best language for new hackers**.
- **Massive Library Support:** Python has libraries like **Scapy (packet manipulation), Requests (web attacks), and Paramiko (SSH attacks)**.
- **Great for Automating Attacks:** You can easily write scripts for **brute-force attacks, phishing, and reconnaissance**.

What Python is Best For:

- **Penetration testing and ethical hacking** (Metasploit, Nmap scripting, etc.).
- **Automating attacks** (brute-force tools, phishing scripts, etc.).
- **Web and network hacking** (exploiting websites, scanning networks, etc.).

Downsides of Python:

- **Slower than Rust, C, and Go**—Python isn't great for real-time exploits or **high-performance tools**.
- **Easier to detect**—Since Python is **interpreted**, security software can analyze and block scripts more easily.

Go (Golang) – The Best for Network Hacking?

Go (Golang) is like **a mix between Python and C**—it's easier to use than C, but much **faster than Python**. Go is **great for networking**, which makes it a powerful language for **building port scanners, brute-force tools, and C2 (command and control) servers**.

Why Go is Good for Hacking:

- **Concurrency Support:** Go has built-in **goroutines**, making it perfect for **handling multiple network requests at once**.
- **Cross-Platform:** Go compiles into a **single executable file**, meaning you can run your hacking tool on **Windows, Linux, and macOS without modifications**.
- **Faster Than Python:** Go is **much faster** than Python, making it great for large-scale attacks.

What Go is Best For:

- **Building port scanners and network tools** (RustScan, masscan alternatives).
- **Developing C2 servers** for red teaming.
- **Writing multithreaded brute-force tools**.

Downsides of Go:

- **Bigger file sizes**—Go binaries are larger than C or Rust executables.
- **Less control over memory—**Not as powerful for exploit development.

C – The Classic Hacking Language

C is like **an old-school lock-picking kit**—it's been used for decades in hacking, and it still works today. C is **super fast and powerful**, giving hackers **direct control over memory**, which is **essential for exploits, malware, and rootkits**.

Why C is Good for Hacking:

- **Speed and Efficiency:** C is one of the **fastest programming languages**, which is great for **high-performance exploits**.
- **Low-Level System Access:** You can **manipulate memory**, **interact with operating system internals**, and **exploit vulnerabilities** directly.
- **Widely Used in Exploit Development:** Many **buffer overflow exploits, shellcode payloads, and privilege escalation tools** are written in C.

What C is Best For:

- **Writing shellcode and exploits**.
- **Developing rootkits and kernel-level malware**.
- **Manipulating system memory for hacking purposes**.

Downsides of C:

- **Memory Management is Hard:** One mistake and your program crashes.
- **Prone to Security Bugs:** C programs are vulnerable to **buffer overflows and memory corruption—**this is why Rust is replacing C for security tools.

Which Language Should You Use for Hacking?

Choose Rust if you want speed and safety together. Rust is great for **advanced hacking tools that are stealthy and hard to detect**, but it's harder to learn.

Choose Python if you want something **easy to use** with **tons of hacking tools** available. It's great for penetration testing, automation, and quick scripting, but it's slower and easier to detect.

Choose Go if you want to **build network scanners, botnets, and C2 servers**. Go is a great mix of **speed and simplicity**, but it's not as powerful for exploit development.

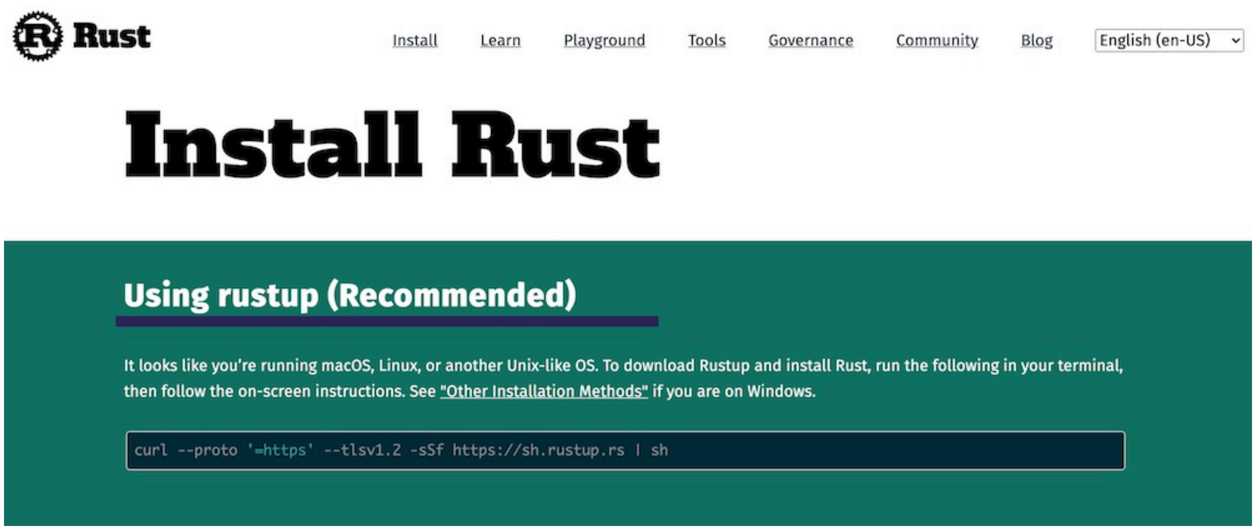
Choose C if you want raw power and complete control over the system. C is the best for **writing exploits, rootkits, and shellcode**, but it's also risky and prone to security bugs.

How to Install Rust (rustup, cargo, rustc) on Linux, Windows, and macOS

Rust is a powerful and safe programming language that is widely used for **system programming, cybersecurity, and hacking tools**. To start using Rust, you need to install three key components:

- **rustup** – The Rust installer and version manager.
- **cargo** – Rust's package manager (like `pip` for Python).
- **rustc** – The Rust compiler that converts Rust code into machine code.

This guide will show you how to install Rust **step-by-step** on **Linux, Windows, and macOS**. Visit their website https://www.rust-lang.org/



Install Rust

Using rustup (Recommended)

It looks like you're running macOS, Linux, or another Unix-like OS. To download Rustup and install Rust, run the following in your terminal, then follow the on-screen instructions. See "Other Installation Methods" if you are on Windows.

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

Figure 1.2 (Image sourced from google)

1. Installing Rust on Windows / Linux

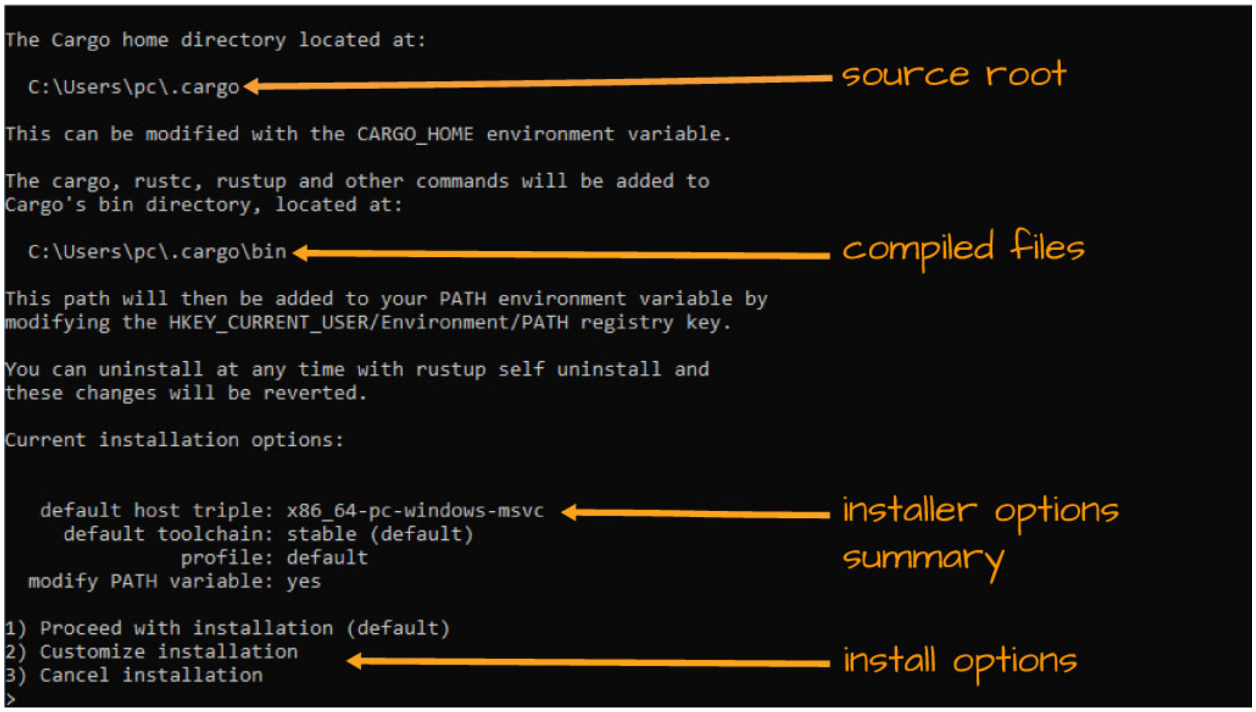
Step 1: Install Rust using rustup

Open a terminal and run the following command:

bash

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

This will download and run the Rust installer



```
The Cargo home directory located at:

  C:\Users\pc\.cargo

This can be modified with the CARGO_HOME environment variable.

The cargo, rustc, rustup and other commands will be added to
Cargo's bin directory, located at:

  C:\Users\pc\.cargo\bin

This path will then be added to your PATH environment variable by
modifying the HKEY_CURRENT_USER/Environment/PATH registry key.

You can uninstall at any time with rustup self uninstall and
these changes will be reverted.

Current installation options:


   default host triple: x86_64-pc-windows-msvc
     default toolchain: stable (default)
               profile: default
  modify PATH variable: yes

1) Proceed with installation (default)
2) Customize installation
3) Cancel installation
>
```

Figure 1.3 (Image sourced from google)

Step 2: Configure Rust Environment

Once installation is complete, restart your terminal and run:

bash

```
source $HOME/.cargo/env
```

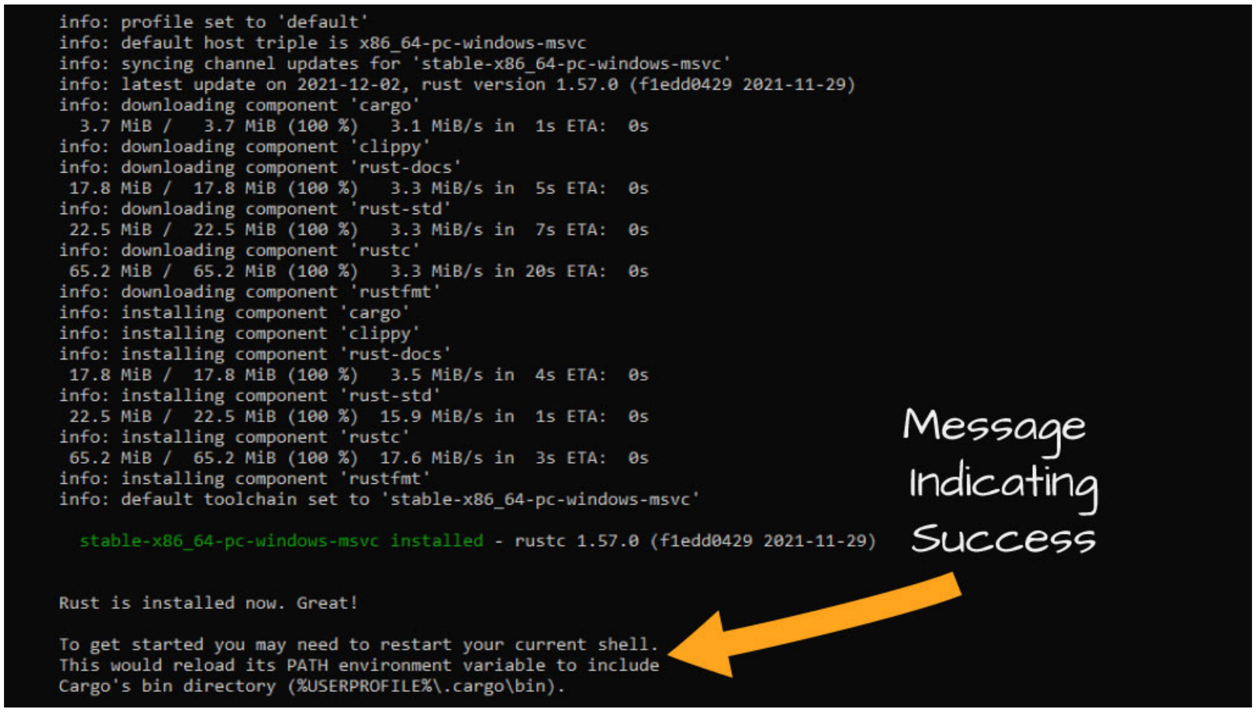
This ensures Rust is properly added to your system's PATH.

Step 3: Verify the Installation

Check if Rust is installed correctly by running:

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

If both commands return a version number, Rust is installed successfully!



```
info: profile set to 'default'
info: default host triple is x86_64-pc-windows-msvc
info: syncing channel updates for 'stable-x86_64-pc-windows-msvc'
info: latest update on 2021-12-02, rust version 1.57.0 (f1edd0429 2021-11-29)
info: downloading component 'cargo'
  3.7 MiB /   3.7 MiB (100 %)   3.1 MiB/s in  1s ETA:  0s
info: downloading component 'clippy'
info: downloading component 'rust-docs'
 17.8 MiB /  17.8 MiB (100 %)   3.3 MiB/s in  5s ETA:  0s
info: downloading component 'rust-std'
 22.5 MiB /  22.5 MiB (100 %)   3.3 MiB/s in  7s ETA:  0s
info: downloading component 'rustc'
 65.2 MiB /  65.2 MiB (100 %)   3.3 MiB/s in 20s ETA:  0s
info: downloading component 'rustfmt'
info: installing component 'cargo'
info: installing component 'clippy'
info: installing component 'rust-docs'
 17.8 MiB /  17.8 MiB (100 %)   3.5 MiB/s in  4s ETA:  0s
info: installing component 'rust-std'
 22.5 MiB /  22.5 MiB (100 %)  15.9 MiB/s in  1s ETA:  0s
info: installing component 'rustc'
 65.2 MiB /  65.2 MiB (100 %)  17.6 MiB/s in  3s ETA:  0s
info: installing component 'rustfmt'
info: default toolchain set to 'stable-x86_64-pc-windows-msvc'

  stable-x86_64-pc-windows-msvc installed - rustc 1.57.0 (f1edd0429 2021-11-29)


Rust is installed now. Great!

To get started you may need to restart your current shell.
This would reload its PATH environment variable to include
Cargo's bin directory (%USERPROFILE%\.cargo\bin).
```

Figure 1.4 (Image sourced from google)

Installing Rust on macOS 🍏

Step 1: Install Rust using rustup

Open **Terminal** and run:

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

Step 2: Configure the Environment

After installation, restart your terminal or run:

bash
```
source $HOME/.cargo/env
```

Step 3: Verify the Installation

Check if Rust and Cargo are installed:

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

Updating and Uninstalling Rust

Updating Rust

Rust updates frequently! To get the latest version, run:

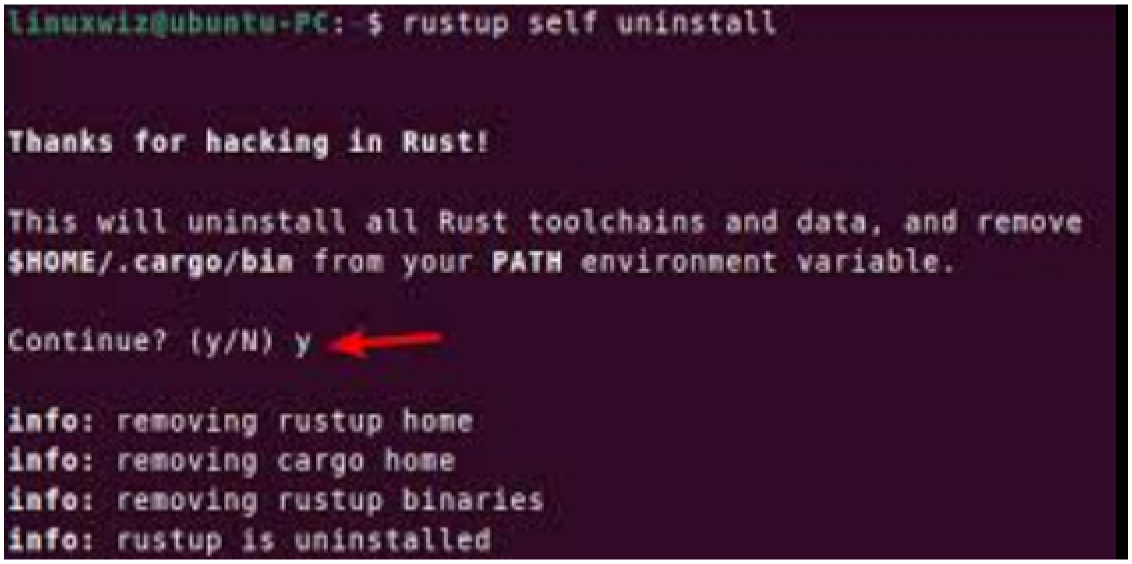
bash
```
rustup update
```

Uninstalling Rust

If you ever need to remove Rust, run:

bash
```
rustup self uninstall
```

This removes Rust and all its components from your system



```
linuxwiz@ubuntu-PC:~$ rustup self uninstall


Thanks for hacking in Rust!

This will uninstall all Rust toolchains and data, and remove
$HOME/.cargo/bin from your PATH environment variable.

Continue? (y/N) y

info: removing rustup home
info: removing cargo home
info: removing rustup binaries
info: rustup is uninstalled
```

Testing Rust with a "Hello, World" Program

To make sure Rust is working properly, create a simple Rust program:

Step 1: Create a New Rust Project

Run the following command:

bash

```
cargo new hello_rust

cd hello_rust
```

This creates a new Rust project in a folder called **hello_rust**.

Step 2: Open the main file

Open the **main.rs** file inside `src/`:

bash

```
nano src/main.rs
```

Replace the existing content with:

rust

```
fn main() {

    println!("Hello, Rust!");

}
```

Step 3: Compile and Run the Program

Run the program using:

bash

```
cargo run
```

If everything is set up correctly, you should see:

bash

```
Hello, Rust!
```

Congratulations! 🎉 Rust is now installed and working on your system. 🚀

Tutorial: Building a Port Scanner in Rust

Figure 1.6 (Image sourced from google)

Reference: Building a Simple Rust-Based Port Scanner

As a hands-on example of using **Rust in offensive security**, let's create a **basic port scanner**. This simple tool will scan for **open ports** on a target machine, giving us a foundational understanding of how Rust handles **network connections and concurrent tasks**.

Step 1: Setting Up the Project

First, create a new Rust project using Cargo:

bash

```
cargo new rust_port_scanner

cd rust_port_scanner
```

Step 2: Writing the Code

}

Open `src/main.rs` and replace its contents with:

rust

```
use std::net::TcpStream;

use std::io::{self};

use std::time::Duration;


fn main() {

    println!("Enter target IP address:");

    let mut target = String::new();

    io::stdin().read_line(&mut target).expect("Failed to read input");

    let target = target.trim();


    println!("Scanning ports on {}", target);


    for port in 1..=1024 {

        let address = format!("{}:{}", target, port);

        let timeout = Duration::from_millis(200);


        if TcpStream::connect_timeout(&address.parse().unwrap(),
timeout).is_ok() {

            println!("Port {} is OPEN", port);

        }

    }
}
```

Step 3: Running the Scanner

Compile and run the program:

bash

```
cargo run
```

Enter a target IP (`127.0.0.1` for localhost), and it will check for **open ports** within the range **1-1024**.

Why This Matters?

This **simple tool** introduces key Rust concepts like **network programming, error handling, and performance optimization**, setting the foundation for more advanced hacking tools. We'll build upon this in later sections, adding **multi-threading, stealth techniques, and evasion tactics** to make the scanner more powerful. 🚀

What's Next?

This example is **just a reference**—a quick demonstration of **what's possible** with Rust. Before we build more **advanced hacking tools**, we will first **deep dive into Rust basics**, including:

- How Rust **manages memory safely** (unlike C, which is prone to buffer overflows).
- Understanding Rust's **ownership and borrowing** system.
- Working with **error handling and concurrency** to build faster, more efficient tools.

Once we've covered these foundational topics, we will **come back to this port scanner** and improve it with **multi-threading, stealth techniques, and advanced scanning methods**. 🚀