**Secure Remote Command Executor**

# 1.Introduction

# The Secure Remote Command Executor is a client-server based application developed in the C programming language. It demonstrates key concepts of low-level networking, process management, and basic security in operating systems. The server continuously listens for incoming TCP connections on a specified port and handles multiple clients concurrently using POSIX threads (pthread).

# Once a client connects, the server performs a basic password-based authentication. Upon successful verification, the client can send Linux shell commands to the server. The server executes these commands using popen() and returns the command output (including errors) back to the client over the secure TCP connection.

# This project emphasizes core systems programming by utilizing raw socket programming in C, I/O redirection for command execution, and multi-threading to support concurrent client sessions. It provides a foundational understanding of secure command execution in distributed systems without relying on external libraries or frameworks, making it highly relevant for educational and systems-level applications

# 2.Objective

The primary objectives of this project are:

1. Develop a secure TCP-based communication channel between a C-based client and server.

2.Implement a basic password-based authentication mechanism to restrict unauthorized access.

3.Enable remote command execution on the server using system-level process handling (popen()).

4.Support concurrent client connections through multi-threaded server architecture using POSIX threads (pthread).

5.Explore fundamental operating system concepts such as:

* Socket programming in C
* Process creation and command execution
* Input/Output redirection to capture command results and errors
* Thread management for concurrent processing
* Secure system interaction through controlled execution of shell commands

**3.System Architecture and Methodology**

**3.1 High level Workflow**

The system follows a classic client-server architecture using TCP sockets for reliable communication. The workflow is as follows::

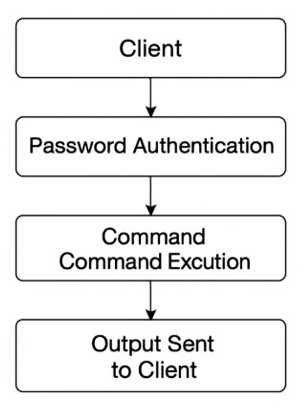
**1. Server:**

* Starts listening on a predefined TCP port for incoming client connections.
* Accepts new connections and spawns a separate thread to handle each client.
* Receives a password from the client and verifies it against a predefined value.
* If authenticated, waits for shell commands from the client.
* Executes the received command securely using popen() to capture both standard output and error output (2>&1).
* Sends the command output back to the client over the same TCP connection.
* Closes the connection when the client sends an exit command.

**2. Client GUI:**

* Connects to the server using TCP sockets.
* Prompts the user to enter a password for authentication.
* Upon successful authentication, allows the user to enter Linux shell commands (e.g., ls, pwd, date).
* Sends the command to the server and waits for the response.
* Displays the server’s response (output or error) in the terminal.

**3.2 Architecture**

****

**3.3 Tools & libraries Used**

|  |  |
| --- | --- |
| **LIBRARY** | **PURPOSE** |
| <socket.h> and <netinet/in.h> | For creating and managing TCP-based client-server communication using sockets. |
| <pthread.h> | Enables multi-threaded handling of multiple client connections on the server side. |
| <stdio.h> and <stdlib.h> | Standard input/output and memory management functions. |
| <string.h> | String manipulation functions for message handling. |
| < arpa/inet.h> | Used for IP address conversions and socket address structures. |
| <unistd.h> | Provides access to POSIX operating system API for low-level system. |

# 4.Implementation Details

**4.1 Source Files**

**1.** **server.c** - Implements a multi-threaded TCP server that listens for client connections, authenticates incoming clients, receives shell commands, executes them using popen(), and sends the command output back to the respective client.

**2.** **client.c** - Implements a terminal-based client that connects to the server, performs password authentication, sends shell commands, receives the output from the server, and displays it on the terminal.

**4.2 Key Functions**

|  |  |
| --- | --- |
| Function | Description |
| socket() | Creates a TCP socket for both client and server communication. |
| bind(), listen(), accept() | Server-side calls to bind to a port, listen for incoming connections, and accept new client requests. |
| connect( ) | Client-side function to connect to the server via TCP. |
| recv(), send() | Executes received shell command and captures the output on the server. |
| pthread\_create() | Spawns a new thread on the server for each connected client to enable concurrent handling. |
| popen() | Executes received shell command on the server and captures standard and error output. |
| fgets() | Reads command output line-by-line from the pipe opened by popen() before sending it back to the client |
| close() | Closes socket descriptors and terminates the connection when the session ends. |
| strncpy(), strstr(), strcspn() | Used for string manipulation such as parsing commands, passwords, and output formatting. |

**4.3 Error Handling**

Invalid Password – The server compares the received password with a hardcoded string. If it does not match, the server responds with "Authentication failed." and closes the connection.

* Socket Connection Failure – If the client fails to connect to the server (e.g., server not running), an error is printed to the terminal using perror().
* Command Execution Errors – If the user sends an invalid command, the server captures error messages via shell redirection (2>&1) and sends the output (including errors) back to the client.
* Server Crash / Client Disconnection – Basic error handling using return values of recv() and send() ensures sockets are closed properly in case of errors or abrupt disconnections.
* Empty Commands – Empty input commands are ignored by the server.
* Large Command Output – Command output is read and sent in chunks using fgets() and send() to handle longer results smoothly.

**4.4 System Calls Used**

* socket() – Creates a TCP socket for client-server communication.
* bind() – Binds the server socket to a specific IP address and port.
* listen() – Sets up the socket to listen for incoming client connections.
* accept() – Accepts an incoming client connection and returns a new socket descriptor for communication.
* connect() – Establishes a connection from the client to the server.
* send() / recv() – Used for transmitting data (passwords, commands, and outputs) between client and server.
* close() – Terminates the socket connection and releases resources.
* popen() – Executes the command in a subprocess and captures the combined stdout and stderr.
* fgets() – Reads command output line-by-line from the process opened by popen().
* perror() – Displays error messages related to system call failures.
* pthread\_create() / pthread\_detach() – Manages concurrent client connections by creating separate threads for each session.

# Testing and Validation

**5.1 Test Environment**

* Platform: Ubuntu 24.04 LTS (also tested under WSL on Windows)
* Language: C (compiled with gcc)
* Client Interface: Command-Line Interface (CLI)
* Communication Protocol: TCP/IP using Berkeley sockets
* Shell: /bin/bash (for executing shell commands)
* Testing Method: Manual functional testing with real command input and output verification.

**5.2 Test Cases**

|  |  |  |
| --- | --- | --- |
| Test case | Description | Expected Result |
| Client connects with correct password | Authenticate with correct password | Authentication Success message from server |
| Client connects with wrong password | Authenticate with wrong password | Authentication Failed message; client disconnected |
| Simple command: ls | List files and folders in current directory | List of files and folders returned to client |
| Command with argument: date | Display system date | **Current system date is retu**rned to client |
| Invalid command: xyz | Send an invalid command | Error message indicating command not found |
| Empty input | Send empty string | Long formatted output returned to client |
| Sleep command: sleep 5 && echo done | Test command that takes time | Server waits for next command; no crash |
| Multiple requests | Send multiple commands in a session | Commands executed one after another with appropriate output |
| cat largefile.txt | Tries to read a file that doesn’t exist | Displays error-No Such file or directory |

**5.3 Test Procedure**

**1. Launch Server**

* Command: ./server
* Expected: Server prints “Server listening on port 8080...” and waits for connections

**2. Launch Client**

* Command: ./client
* Expected: Prompts user for password in terminal

**3. Authenticate Client**

* Input: Enter correct password (admin123)
* Expected: Server sends "Authenticated. You may now send commands." to client

**4. Run Functional Commands**

* Input: Shell commands like ls, pwd, whoami
* Expected: Output is displayed in client terminal

**5. Error Handling**

* Input: Invalid command like foobar
* Expected: Server captures shell error and returns to client

**6. Command with Delay**

* Input: sleep 3 && echo Hello
* Expected: “Hello” appears after a delay without freezing server/client

**7. Shutdown Handling**

* Action: Stop server manually (e.g., Ctrl+C)
* Input: Client tries to send command
* Expected: Client shows connection closed / error message

**8. GUI Responsiveness**

* Action: Send several commands one after another
* Expected: Each command is processed independently; no crash or hang

# Results

* All valid commands were successfully executed after proper authentication.
* Incorrect passwords were correctly identified and connections were securely denied.
* Command outputs, including long and multi-line results, were accurately transmitted to the client.
* The multi-threaded server handled multiple client connections efficiently without crashing.
* Client and server terminated gracefully upon exit, even after abrupt disconnection scenarios.

# Conclusion

The project demonstrates a practical application of low-level socket programming, multi-threading, and secure command execution using the C programming language. By combining TCP-based client-server architecture with basic authentication and process management, this project simulates the core functionality of a remote system administration tool. It serves as a strong foundational project for understanding operating system-level concepts and networked application development in C.

# Learning Outcomes

1. Hands-on experience with socket programming in C.   
2. Understanding of multi-threaded server design using POSIX threads.  
3. Implementation of TCP client-server communication using Berkeley sockets.  
4. Use of command execution via popen() and secure redirection of standard/error output. 5. Exposure to Linux shell integration and system-level programming using C.

# References

Linux Manual Pages: https://man7.org/linux/man-pages/

Beej’s Guide to Network Programming: https://beej.us/guide/bgnet/

GNU C Library Documentation: https://www.gnu.org/software/libc/manual/

POSIX Thread Programming: https://man7.org/linux/man-pages/man7/pthreads.7.html