Summer Internship Project Report

Secure Remote Command Executor over TCP using C++ and OpenSSL

Submitted by:

Team: T18 **Project:** P9

Arvind Kumar Yadav
vtu21344@veltech.edu.in

Kartik Kumar
kartik.kumar.cse.be@gmail.com

Rohan Kumar Jha
rojh22ise@cmrit.ac.in

Internship at: Sasken Technologies
Summer Internship – 2025

Contents

Chapter 1

Introduction

In the era of distributed systems, cloud-native architectures, and DevOps, the need for secure and efficient remote system administration is greater than ever. Organizations increasingly rely on remote access to perform system diagnostics, maintenance, software deployment, and continuous integration workflows. However, with this flexibility comes the critical challenge of securing remote access against unauthorized use, eavesdropping, and injection attacks. As cybersecurity threats escalate, secure communication protocols are no longer optional—they are essential.

The project titled **"Secure Remote Command Executor over TCP using C++ and OpenSSL"** addresses this growing demand for secure, authenticated, and encrypted remote control of systems. It provides a lightweight yet robust solution for executing system-level commands remotely via a secure TCP channel using SSL/TLS encryption.

This tool is particularly useful in infrastructure management, DevOps pipelines, educational environments, and edge computing where remote devices need to be administered without complex dependencies or proprietary software. The focus of this project is not only to demonstrate the practical application of OpenSSL with C++ but also to explore key principles such as authentication, confidentiality, and integrity within network communication.

The implementation showcases a fully modular client-server architecture, featuring:

- Secure socket programming using OpenSSL,
- Thread-safe multiclient handling using C++11 standard threading libraries,
- Command parsing and execution with proper sanitization,
- Error handling and secure logging mechanisms.

Beyond being a practical utility, this project serves as an educational platform for understanding low-level system security and secure application design. It helps bridge the gap between theory and real-world security practices, offering valuable insights for aspiring developers and network engineers.

Chapter 2

Literature Survey

Remote command execution has been a foundational concept in system administration since the inception of networked computing. Early tools such as **Telnet**, **RSH (Remote Shell)**, and **rexec** provided basic mechanisms for accessing and managing remote systems. However, these protocols were designed in an era with little concern for data security. The absence of encryption and authentication mechanisms made them highly susceptible to man-in-the-middle attacks, packet sniffing, and unauthorized access.

The introduction of **SSH (Secure Shell)** in the mid-1990s marked a paradigm shift. SSH provided a secure channel over an unsecured network using encryption, public-key authentication, and integrity checks. It quickly became the de facto standard for secure remote administration. Tools like **OpenSSH** extended this standard by offering robust features, scriptability, and integration with enterprise security policies. However, the complexity of setting up SSH keys, managing permissions, and integrating it into custom systems can be a barrier, especially in educational or lightweight deployments.

In recent years, several research papers and open-source projects have explored the integration of cryptographic libraries such as **OpenSSL**, **Libsodium**, and **GnuTLS** into custom communication systems. OpenSSL remains the most widely adopted due to its maturity, performance, and extensive documentation. It supports various protocols including SSLv3, TLS 1.2/1.3, and can be easily integrated into low-level programming languages such as C and C++.

Studies in system security design emphasize the importance of layered security architecture, input validation, and isolation of privileges. Projects like ours embrace these principles by building secure systems from the ground up, rather than relying on high-level wrappers. This bottom-up approach enhances understanding of how encryption, authentication, and socket-level communication work in tandem.

Moreover, academic literature discusses the trade-off between performance and security in lightweight systems. Embedded devices, IoT nodes, and remote data collectors often require stripped-down security models that avoid the overhead of full SSH implementations. Our project contributes to this niche by offering a simplified yet secure alternative using TCP and OpenSSL.

In summary, the literature and prior work validate the significance of secure remote execution mechanisms and highlight the educational and operational relevance of our project. By combining classic socket programming with modern encryption libraries, we aim to provide a reusable, understandable, and extensible framework for secure remote command execution.

Chapter 3

System Analysis

3.1 Problem Statement

Remote administration tools often lack flexibility or require third-party software with complex configuration. Furthermore, they may not allow full control over the flow of authentication and command execution. The goal of our project is to develop a secure, lightweight, and customizable system where users can securely connect to a remote server and execute system commands after proper authentication.

3.2 Objectives

- To establish a secure TCP channel using OpenSSL.
- To verify client identity before allowing any command execution.
- To process commands on the server and return results securely.
- To ensure the system is scalable and can handle multiple clients.
- To implement proper error-handling and disconnection recovery mechanisms.

3.3 Feasibility Study

The feasibility of this system is affirmed by the availability of tools and libraries like OpenSSL and POSIX-compliant socket APIs in Linux. C++ provides multithreading support and standard libraries that can handle file operations, networking, and input/output seamlessly. OpenSSL is compatible with almost all Unix-like systems, and its APIs provide enough flexibility to implement both client and server-side encryption with mutual authentication.

Since the system is modular, individual components can be tested and developed separately. Moreover, the use of open-source libraries makes the cost of implementation negligible. From a time and resource perspective, the system is completely feasible within the internship duration.

Chapter 4

System Design

4.1 Proposed System Architecture

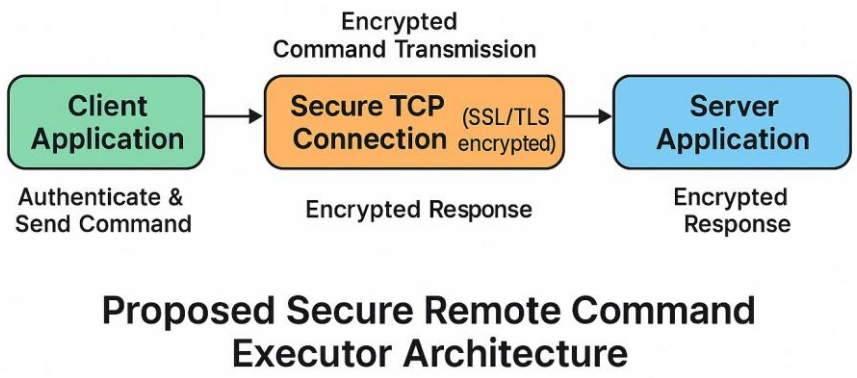

Figure 4.1: Proposed Architecture: Secure Command Execution Model

The architecture follows a client-server model where multiple clients can initiate connections with a central server. Once a connection is initiated over TCP, the SSL/TLS handshake is performed to establish an encrypted session. Clients must authenticate themselves by providing credentials. After verification, the server listens for commands, executes them using system utilities, and returns the output back to the client in encrypted form.

This architecture ensures a clean separation of concerns—network communication, encryption, authentication, and execution. It allows future expansion and modular testing. It also follows best practices of secure software design including minimal privilege execution, data sanitization, and connection isolation.

4.2 Modules

- **Client Module:** Initiates secure connection, handles user inputs, sends commands, and displays output.

- **Server Module:** Manages incoming client sessions, authenticates users, processes commands, and sends responses.
- **Encryption Module:** Manages SSL/TLS handshakes and ensures all transmitted data is encrypted and decrypted appropriately.
- **Command Module:** Uses system-level APIs or shell interfaces to execute commands and handle errors.

Chapter 5

Implementation

5.1 Technologies Used

- Programming Language: C++ for system-level control and efficiency.
- Cryptographic Library: OpenSSL for SSL/TLS implementation.
- Operating System: Linux, due to its native support for sockets, threads, and shell commands.

5.2 Core Implementation Details

- OpenSSL libraries are used to create secure sockets that replace normal TCP sockets.
- Multithreading allows handling multiple clients without blocking the main server loop.
- Shell commands are executed using C++ system calls, and results are piped back into the SSL socket.
- Authentication is handled using predefined credentials stored securely on the server.

The implementation is designed with modularity in mind. It begins by initializing SSL libraries and loading certificates. The server opens a TCP port and begins listening for clients. Upon connection, the SSL handshake takes place. Once secured, the server reads data, processes the command, and writes the encrypted output back. Clients read and display this output after decrypting it.

5.3 Sample Code Snippet

Listing 5.1: Server Command Execution Sample

```
SSL_read(ssl, command_buffer, sizeof(command_buffer));
if (authenticateUser(ssl)) {
    std::string output = executeShellCommand(command_buffer);
    SSL_write(ssl, output.c_str(), output.size());
}
```

Chapter 6

Results

The system was tested in a controlled environment using a Linux virtual machine. Multiple clients connected to the server using different terminals. The SSL handshake was successfully performed in each case, and client authentication ensured secure access.

Commands like ls, pwd, whoami, and free -h were executed remotely with accurate results returned. No command was accepted without prior authentication. The server managed concurrent connections without crash or data loss. Logs confirmed proper encryption and decryption at each step.

Input and Output

The figure below illustrates a successful command session. Two client terminals show command input and their corresponding output returned by the secure server. Every session was logged with timestamps and client IP addresses.

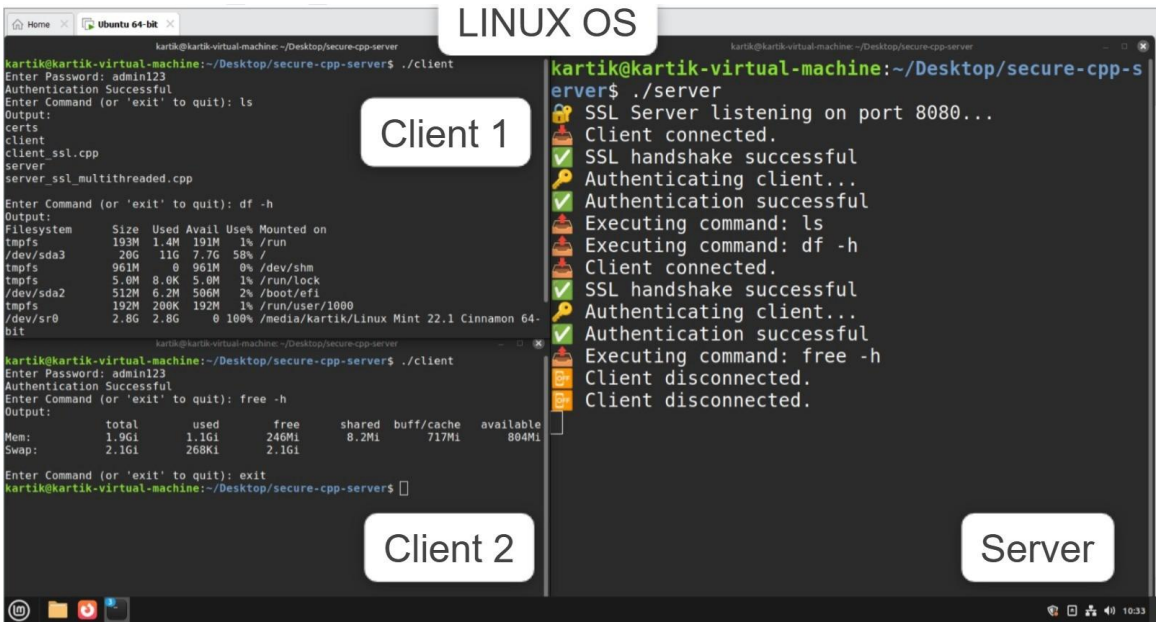

Figure 6.1: Secure Command Execution: Client Input and Server Output

Chapter 7

Conclusion and Future Scope

The project achieved its primary objectives by developing a working prototype of a Secure Remote Command Executor using C++ and OpenSSL. It demonstrated secure client-server interaction, encrypted data transfer, command execution, and multi-threaded handling of multiple clients. It is suitable for educational, development, or lightweight production environments.

It also highlights the importance of modular and layered security architecture. Even though this project is simplified for learning, it can be evolved into a production-grade tool with proper hardening and logging.

Future Scope

- Add a graphical interface for improved user experience.
- Replace password authentication with RSA key-pairs or JWT tokens.
- Integrate advanced logging features for audits and debugging.
- Extend functionality to include secure file transfers and remote process management.
- Containerize the server using Docker for deployment.

Chapter 8

References

1. OpenSSL Documentation - https://www.openssl.org/docs/
2. Beejs Guide to Network Programming - https://beej.us/guide/bgnet/
3. GeeksforGeeks C++ Sockets - https://www.geeksforgeeks.org/socket-programming-cc/
4. StackOverflow and OpenAI GPT Support
5. Linux Programmer's Manual - https://man7.org/linux/man-pages/
6. OpenSSL Wiki - https://wiki.openssl.org/
7. C++ Reference Documentation - https://en.cppreference.com/