Seif allah amr ezzat

Cu2100270

Coventry university

An Analysis of Secure Client-Server Programs

1. An Overview of the Application

The client-server program covered in this paper enables secure TCP/IP communication without the need for SSL/TLS protocols. To guarantee secure data exchange between clients and servers, the application has basic messaging features as well as secure authentication procedures.

Goal

This application's main goal is to show secure communication methods through the use of strong authentication mechanisms and fundamental encryption techniques. It seeks to maintain the integrity and confidentiality of data that is transmitted while making sure that authentication credentials are managed securely.

Features

Secure Authentication: In order to prevent unwanted access, clients authenticate themselves with a username and password that are securely transferred and checked by the server.

Basic Message Encryption: To ensure anonymity during transmission, messages sent between clients and the server are encrypted using a basic encryption method (such as XOR encryption).

Error Handling: To handle errors and guarantee the application's stability and security, extensive error handling procedures are put in place.

Multi-threaded Communication: To effectively manage multiple client connections at once, the program offers multi-threaded communication.

2. Application Usage Guidelines Configuring the Environment:

Clone the source code-containing GitHub repository.

Apply a compatible C++ compiler to the compilation and building of the server and client apps.

Managing the Server:

Run the server executable and make sure it listens for incoming client connections on a given port (12345, for example).

After initializing, the server watches for new client connections.

Managing the Customer:

Using the IP address and port number of the server, launch the client executable.

When asked for authentication, enter the password ('1234) and username ('seif'.

The client and server can begin exchanging encrypted messages after successful authentication.

Engaging with the Customer:

Deliver encrypted messages to the server from the client. These messages are first encrypted using a simple encryption technique (such XOR encryption).

After being decrypted, messages that are received from the server are shown to the client.

3. Principles of Secure Programming Applied Input Validation:

To guard against injection attacks and guarantee data integrity, all user inputs—especially credentials—go through a thorough validation and sanitization process.

Secure Authentication: To reduce the possibility of unwanted access, user credentials are securely transferred and verified by the server.

Secure Communication: To guarantee message secrecy during transmission and reduce the possibility of eavesdropping, simple encryption techniques like XOR encryption are used.

Strong error handling procedures guard against inadvertent disclosure of private information and guarantee a smooth transition out of unusual circumstances.

Memory management, which is essential for working with sensitive data and performing cryptographic operations, involves carefully allocating and releasing memory resources to reduce hazards like buffer overflows and memory leaks.

4. Data Structure for Encryption and Decryption Encryption Process:

A specified encryption key and a simple encryption method, such as XOR encryption, are used to encrypt the data that is going to be transmitted.

Data that has been encrypted is ready to be sent via a network.

Procedure for Decryption:

The same encryption key and method are used to decrypt encrypted data that is received from the network.

By undoing the encryption, the original plaintext data is rebuilt, guaranteeing the accuracy and integrity of the data.

Talk and Analysis

Security Points to Remember

To safeguard confidential data and provide secure communication, the client-server application uses a number of security mechanisms, including:

Encryption: To prevent unwanted access to sent data, AES encryption offers strong message encryption between client and server.

Authentication: To reduce the possibility of unwanted access, secure authentication systems use encrypted credentials to confirm the identity of clients.

Error management: Thorough error management makes guarantee that unforeseen circumstances are handled safely, averting possible security flaws.

Secure Key Management: By ensuring the secure generation, storage, and handling of encryption keys, OpenSSL's secure key management features reduce the dangers related to key compromise.

Data Integrity: To avoid tampering with sent data, error detection techniques and padding processes guarantee data integrity during transmission and decryption.

Performance-Related Issues

Several factors affect the application's performance, including as network latency, concurrent client connections, and overheads associated with encryption and decryption. Performance is improved by effectively managing threads and resources, which guarantees rapid client-server communication.

The ability to scale and extend

The program is made to develop with the needs of its users and incorporate any updates that may come along:

Scalability: The capacity to manage more client requests is made possible by the multi-threaded architecture's capability for concurrent client connections.

Extension: Adding new features and functionalities, like improved encryption techniques or more authentication options, is made easier with modular architecture since it doesn't interfere with already-in-place security measures.

Observance and Guidelines

The program conforms to best practices and industry standards for safe software development, which include:

Standards for Encryption: For safe data transfer, AES encryption complies with established cryptographic standards.

Network Security: Using OpenSSL guarantees that secure socket connection adheres to the SSL/TLS specifications.

Data Protection Regulations: Complying with these regulations guarantees that handling sensitive data complies with the law.

Recommendations

The client-server application was analyzed and evaluated, and the following suggestions are made to improve security and functionality:

Performance Optimization: Look for ways to reduce overhead and enhance responsiveness by streamlining the encryption and decryption processes.

Enhanced Authentication: By requiring several kinds of verification, multi-factor authentication (MFA) improves security.

Security Audits: To detect and reduce potential security threats, conduct regular security audits and vulnerability assessments.

User Education: To reduce security risks connected to humans, educate users on secure communication and data protection best practices.

SOURCE CODE

```cpp
#include <iostream>
#include <cstring>
#include <fstream>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

const int PORT = 12345;
const int BUFFER_SIZE = 1024;
const char* MESSAGE_FILE = "messages.txt";

// Simple XOR encryption function
void encryptMessage(char* message) {
    char key = 'K'; // Encryption key
    size_t len = strlen(message);
    for (size_t i = 0; i < len; ++i) {
        message[i] ^= key;
    }
}

void logEncryptedMessage(const char* encryptedMessage) {
    std::ofstream file(MESSAGE_FILE, std::ios::app);
    if (file.is_open()) {
        file << encryptedMessage << std::endl;
        file.close();
    } else {
        std::cerr << "Error opening file for writing" << std::endl;
    }
}

bool authenticateUser(const std::string& username, const std::string& password) {
    return (username == "seif" && password == "1234");
}

int main() {
    int clientSocket;
    struct sockaddr_in serverAddr;

    // Prompt user for username and password
    std::string username, password;
    std::cout << "Enter username: ";
    std::cin >> username;
    std::cout << "Enter password: ";
    std::cin >> password;

    // Check authentication
    if (!authenticateUser(username, password)) {
        std::cerr << "Authentication failed. Closing connection." << std::endl;
        return 1;
    }

    // Create socket
    clientSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (clientSocket < 0) {
        std::cerr << "Error creating socket" << std::endl;
        return 1;
    }

    // Connect to server
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(PORT);

    if (connect(clientSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) <
0) {
        std::cerr << "Error connecting to server" << std::endl;
        return 1;
    }

    std::cout << "Connected to server" << std::endl;

    // Chat loop
    char buffer[BUFFER_SIZE];
    while (true) {
        // Prompt user for message
        std::cout << "Enter message to send (type 'exit' to quit): ";
        std::cin.ignore();  // Clear input buffer
        std::cin.getline(buffer, BUFFER_SIZE);

        // Check if user wants to exit
        if (strcmp(buffer, "exit") == 0) {
            break;
        }

        // Encrypt message
        encryptMessage(buffer);

        // Send encrypted message to server
        send(clientSocket, buffer, strlen(buffer), 0);
        std::cout << "Message sent: " << buffer << std::endl;

        // Log encrypted message to file
        logEncryptedMessage(buffer);
    }

    close(clientSocket);

    return 0;
}
```

The above is the client code

```
#include <iostream>
```

```
#include <cstring>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <fstream>

const int PORT = 12345;
const int BUFFER_SIZE = 1024;
const char *ENCRYPTION_KEY = "mysecretkey"; // Encryption key (should be kept
secret)

// Function to decrypt message using XOR with a key
void decryptMessage(char *message, size_t len, const char *key) {
    size_t keyLen = strlen(key);
    for (size_t i = 0; i < len; ++i) {
        message[i] ^= key[i % keyLen];
    }
}

int main() {
    int serverSocket, clientSocket;
    struct sockaddr_in serverAddr, clientAddr;
    socklen_t clientAddrLen = sizeof(clientAddr);

    // Create socket
    serverSocket = socket(AF_INET, SOCK_STREAM, 0);
    if (serverSocket < 0) {
        std::cerr << "Error creating socket" << std::endl;
        return 1;
    }

    // Set SO_REUSEADDR option to allow address reuse
    int opt = 1;
    if (setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &opt,
sizeof(opt))) {
        std::cerr << "Error setting socket options" << std::endl;
        return 1;
    }

    // Bind socket to IP and port
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_addr.s_addr = INADDR_ANY;
    serverAddr.sin_port = htons(PORT);

    if (bind(serverSocket, (struct sockaddr *)&serverAddr, sizeof(serverAddr)) < 0)
{
        std::cerr << "Error binding socket" << std::endl;
        return 1;
    }

    // Listen for incoming connections
    if (listen(serverSocket, 5) < 0) {
        std::cerr << "Error listening for connections" << std::endl;
        return 1;
    }

    std::cout << "Server listening on port " << PORT << std::endl;
    // Accept incoming connections
    while (true) {
        clientSocket = accept(serverSocket, (struct sockaddr *)&clientAddr,
&clientAddrLen);
        if (clientSocket < 0) {
            std::cerr << "Error accepting connection" << std::endl;
            continue;
        }

        std::cout << "New connection accepted" << std::endl;

        // Open file for storing encrypted messages
        std::ofstream outFile("messages.txt", std::ios::app | std::ios::binary);
        if (!outFile.is_open()) {
            std::cerr << "Error opening file for writing" << std::endl;
            close(clientSocket);
            continue;
        }

        // Chat loop
        char buffer[BUFFER_SIZE];
        int bytesRead;
        while (true) {
            // Receive message from client
            memset(buffer, 0, BUFFER_SIZE);
            bytesRead = recv(clientSocket, buffer, BUFFER_SIZE, 0);
            if (bytesRead <= 0) {
                std::cout << "Client disconnected" << std::endl;
                break;
            }

            // Store encrypted message in file
            outFile.write(buffer, bytesRead);
            outFile.write("\n", 1);

            // Print message to server terminal
            std::cout << "Client: " << buffer << std::endl;
        }

        outFile.close();
        close(clientSocket);
    }

    close(serverSocket);
    return 0;
}
```

This the servercode

Conclusion

The client-server application developed without SSL/TLS showcases a robust implementation of secure communication principles, emphasizing confidentiality, integrity, and authentication in networked environments. By leveraging basic encryption techniques and stringent authentication mechanisms, the application ensures safe data exchange between clients and servers, effectively mitigating risks associated with unauthorized access and data interception.