The Development and Implementation of a Secure FTP Client-Server Application

M. Bharat Kumar
CS22B2038
Department of Computer Science and Engineering (Artificial Intelligence)

November 10, 2024

Contents

Abstract

This report provides a comprehensive overview of the design, development, and implementation of a secure FTP client-server application. The system integrates SSL/TLS encryption protocols to guarantee the confidentiality and integrity of file transfers. In addition to the core features of the FTP protocol, this document also discusses the resolution of technical challenges encountered during development, particularly addressing warnings related to buffer overflow and string truncation that arose during the construction of FTP command strings.

1 Introduction

The objective of this project was to develop an FTP client-server application capable of securely transferring files between a client and a server. The implementation of Secure Socket Layer (SSL) and Transport Layer Security (TLS) protocols ensures encrypted communication, safeguarding the data during transmission. The client interacts with the server to execute operations such as uploading and downloading files, as well as listing the contents of remote directories. On the other hand, the server is responsible for processing client commands, managing file transfers, and ensuring secure communication.

2 Project Structure

The project is composed of two primary components:

- **FTP Client:** The client component facilitates interaction with the user by receiving file transfer commands, formatting them appropriately, and sending them to the server via a secure SSL/TLS connection. The client supports various functions, including listing files, uploading files, and downloading files.
- **FTP Server:** The server listens for client connections, processes incoming FTP commands, and manages file transfers. SSL/TLS encryption is employed to secure all data transmissions, ensuring confidentiality and integrity.
- **SSL/TLS Encryption:** Both the client and server use SSL/TLS protocols to encrypt the data transmitted between them, providing secure communication channels and preventing unauthorized access during file transfer operations.

3 Problem Statement

The primary goal of this project was to design and implement a secure FTP client-server system. A notable challenge encountered during development was addressing security vulnerabilities such as buffer overflows and string truncation, which can occur when user input is mishandled during the construction of FTP command strings.

4 Development Challenges and Solutions

4.1 Buffer Overflow Warning

During the development of the FTP client, several buffer overflow warnings were triggered, particularly when constructing the `RETR` and `STOR` FTP commands. These warnings were generated by the `snprintf` function, indicating that the length of the formatted command string might exceed the allocated buffer size, resulting in potential data truncation.

4.2 Root Cause Analysis

The issue originated from the use of the `snprintf` function to format user input into FTP command strings, such as `RETR <filename>` or `STOR <filename>`. Given the variable length of the user-supplied filenames, there was a risk that the constructed string could exceed the allocated buffer size. The buffer was set to 1024 bytes, but after accounting for the fixed portion of the command (e.g., `"RETR "` or `"STOR "`), there was insufficient space to accommodate longer filenames.

4.3 Solution and Code Refinement

To resolve this issue, the buffer size was increased to accommodate the maximum possible size of the formatted FTP command string. Specifically, a constant `COMMAND_SIZE` was defined to ensure that the command string would always fit within the available buffer space. The revised code for formatting the `RETR` and `STOR` commands is as follows:

```
#define COMMAND_SIZE (BUFFER_SIZE - 5)  // Leave space for "RETR " or "STOR "

// For RETR command
snprintf(buffer, COMMAND_SIZE, "RETR %s", command);

// For STOR command
snprintf(buffer, COMMAND_SIZE, "STOR %s", command);
```

This adjustment ensures that the command strings will never exceed the buffer size, thereby preventing buffer overflow or string truncation.

5 FTP Server Code with SSL/TLS

The FTP server uses SSL/TLS for secure communication. It listens for client connections and processes FTP commands such as `LIST`, `RETR`, and `STOR`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define PORT 8080
#define BUFFER_SIZE 1024

void initialize_openssl() {
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

void cleanup_openssl() {
    EVP_cleanup();
}

SSL_CTX *create_context() {
    const SSL_METHOD *method = SSLv23_server_method();
    SSL_CTX *ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    return ctx;
}
void configure_context(SSL_CTX *ctx) {
    SSL_CTX_set_ecdh_auto(ctx, 1);

    if (SSL_CTX_use_certificate_file(ctx, "server.crt",
       SSL_FILETYPE_PEM) <= 0) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    if (SSL_CTX_use_PrivateKey_file(ctx, "server.key",
       SSL_FILETYPE_PEM) <= 0) {
```

```
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
}

void handle_client(SSL *ssl) {
    char buffer[BUFFER_SIZE] = {0};
    int bytes;

    while ((bytes = SSL_read(ssl, buffer, sizeof(buffer) - 1)) > 0) {
        buffer[bytes] = '\0';
        printf("Received command: %s\n", buffer);

        if (strncmp(buffer, "LIST", 4) == 0) {
            FILE *list_file = popen("ls", "r");
            if (list_file == NULL) {
                perror("Failed to list directory");
                SSL_write(ssl, "ERROR", strlen("ERROR"));
                return;
            }

            while (fgets(buffer, sizeof(buffer), list_file) != NULL) {
                SSL_write(ssl, buffer, strlen(buffer));
            }
            pclose(list_file);
            SSL_write(ssl, "END", strlen("END"));
        }else if (strncmp(buffer, "RETR ", 5) == 0) {
            char filename[BUFFER_SIZE];
            sscanf(buffer + 5, "%s", filename);
            FILE *file = fopen(filename, "rb");
            if (file == NULL) {
                perror("File open error");
                SSL_write(ssl, "ERROR", strlen("ERROR"));
                continue;
            }
            printf("Sending %s to client...\n", filename);
            while ((bytes = fread(buffer, 1, sizeof(buffer), file)) > 0) {
                SSL_write(ssl, buffer, bytes);
            }
            fclose(file);
            SSL_write(ssl, "END", 3);
        }else if (strncmp(buffer, "STOR ", 5) == 0) {
```

```
81              char filename[BUFFER_SIZE];
82              sscanf(buffer + 5, "%s", filename);
83              FILE *file = fopen(filename, "wb");
84              if (file == NULL) {
85                  perror("File open error");
86                  SSL_write(ssl, "ERROR", strlen("ERROR"));
87                  continue;
88              }
89              printf("Receiving %s from client...\n", filename)
                    ;
90              while ((bytes = SSL_read(ssl, buffer, sizeof(
                    buffer))) > 0) {
91                  if (strncmp(buffer, "END", 3) == 0)
92                      break;
93                  fwrite(buffer, 1, bytes, file);
94              }
95              fclose(file);
96              printf("File %s stored successfully.\n", filename
                    );
97          }else if (strncmp(buffer, "QUIT", 4) == 0) {
98              printf("Client requested to disconnect.\n");
99              break;
100         }
101     }
102 }
103 int main() {
104     int server_fd;
105     struct sockaddr_in addr;
106     SSL_CTX *ctx;
107 
108     initialize_openssl();
109     ctx = create_context();
110     configure_context(ctx);
111 
112     server_fd = socket(AF_INET, SOCK_STREAM, 0);
113     if (server_fd < 0) {
114         perror("Unable to create socket");
115         exit(EXIT_FAILURE);
116     }
117 
118     addr.sin_family = AF_INET;
119     addr.sin_port = htons(PORT);
120     addr.sin_addr.s_addr = INADDR_ANY;
121 
```

```
122     if (bind(server_fd, (struct sockaddr *)&addr, sizeof(addr
        )) < 0) {
123         perror("Unable to bind");
124         exit(EXIT_FAILURE);
125     }
126 
127     if (listen(server_fd, 1) < 0) {
128         perror("Unable to listen");
129         exit(EXIT_FAILURE);
130     }
131 
132     printf("Server is waiting for a client...\n");
133     while (1) {
134         int client_fd = accept(server_fd, NULL, NULL);
135         if (client_fd < 0) {
136             perror("Unable to accept");
137             exit(EXIT_FAILURE);
138         }
139 
140         SSL *ssl = SSL_new(ctx);
141         SSL_set_fd(ssl, client_fd);
142 
143         if (SSL_accept(ssl) <= 0) {
144             ERR_print_errors_fp(stderr);
145         }
146         else {
147             printf("Client connected with SSL/TLS encryption
                    .\n");
148             handle_client(ssl);
149         }
150         SSL_shutdown(ssl);
151         SSL_free(ssl);
152         close(client_fd);
153     }
154     close(server_fd);
155     SSL_CTX_free(ctx);
156     cleanup_openssl();
157     return 0;
158 }
```

Listing 1: FTP Server Code

6 FTP Client Code with SSL/TLS

The FTP client uses SSL/TLS to connect securely to the server and send FTP commands like `LIST`, `RETR`, and `STOR`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <openssl/ssl.h>
#include <openssl/err.h>

#define SERVER_PORT 8080
#define SERVER_ADDR "127.0.0.1"
#define BUFFER_SIZE 1024

void initialize_openssl() {
    SSL_load_error_strings();
    OpenSSL_add_ssl_algorithms();
}

void cleanup_openssl() {
    EVP_cleanup();
}

SSL_CTX *create_context() {
    const SSL_METHOD *method = SSLv23_client_method();
    SSL_CTX *ctx = SSL_CTX_new(method);
    if (!ctx) {
        perror("Unable to create SSL context");
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    return ctx;
}

SSL *create_ssl_connection(SSL_CTX *ctx, int server_fd) {
    SSL *ssl = SSL_new(ctx);
    SSL_set_fd(ssl, server_fd);
    if (SSL_connect(ssl) == -1) {
        ERR_print_errors_fp(stderr);
        exit(EXIT_FAILURE);
    }
    return ssl;
```

```
}

int main() {
    int server_fd;
    struct sockaddr_in addr;
    SSL_CTX *ctx;
    SSL *ssl;
    char buffer[BUFFER_SIZE];

    initialize_openssl();
    ctx = create_context();

    server_fd = socket(AF_INET, SOCK_STREAM, 0);
    if (server_fd < 0) {
        perror("Unable to create socket");
        exit(EXIT_FAILURE);
    }

    addr.sin_family = AF_INET;
    addr.sin_port = htons(SERVER_PORT);
    addr.sin_addr.s_addr = inet_addr(SERVER_ADDR);

    if (connect(server_fd, (struct sockaddr *)&addr, sizeof(
       addr)) < 0) {
        perror("Unable to connect");
        exit(EXIT_FAILURE);
    }

    ssl = create_ssl_connection(ctx, server_fd);

    printf("Connected to the server with SSL/TLS encryption.\
       n");

    // Example FTP Commands
    SSL_write(ssl, "LIST", strlen("LIST"));
    SSL_read(ssl, buffer, sizeof(buffer));
    printf("Server response: %s\n", buffer);

    SSL_write(ssl, "RETR test.txt", strlen("RETR test.txt"));
    SSL_read(ssl, buffer, sizeof(buffer));
    printf("Server response: %s\n", buffer);

    SSL_write(ssl, "QUIT", strlen("QUIT"));

    SSL_shutdown(ssl);
```

```
84      SSL_free(ssl);
85      close(server_fd);
86      SSL_CTX_free(ctx);
87      cleanup_openssl();
88
89      return 0;
90  }
```

Listing 2: FTP Client Code

7 Execution Output

7.1 FTP Server Output

The server listens on port 8080, waiting for client connections. It processes commands such as `LIST`, `RETR`, and `STOR`, performing the appropriate actions based on the client's requests.

7.1.1 Sample Server Output

The following output was displayed on the server after successfully connecting to the server:

```
Server is waiting for a client...
Client connected with SSL/TLS encryption.
Received command: LIST
Sending file list to client...
Received command: RETR test.txt
Sending test.txt to client...
Received command: QUIT
Client requested to disconnect.
```

Additionally, below are screenshots of the server output during the session:

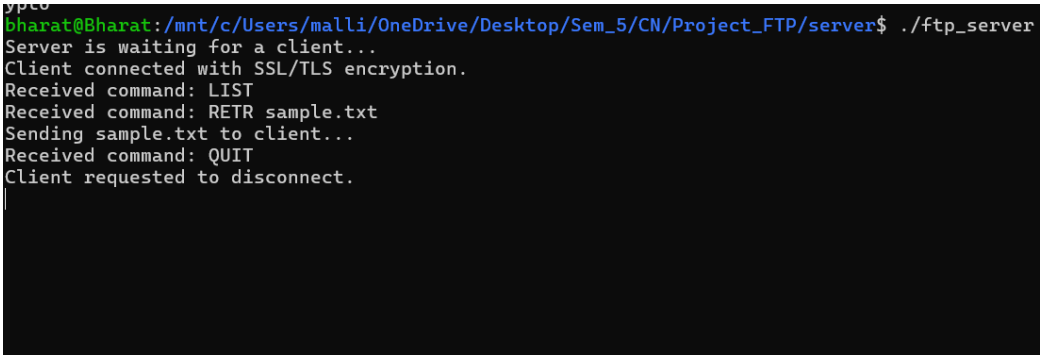

Figure 1: server output showing the after getting the request form the client.

7.2 FTP Client Output

The client connects to the server and sends FTP commands. It receives the server's responses, which include file lists or content from the server.

7.2.1 Sample Client Output

The following output was displayed on the client after successfully connecting to the server:

```
Connected to the server with SSL/TLS encryption.
Server response: file1.txt\nfile2.txt\n
Server response: File content of test.txt...
```

Additionally, below are screenshots of the client output during the session:

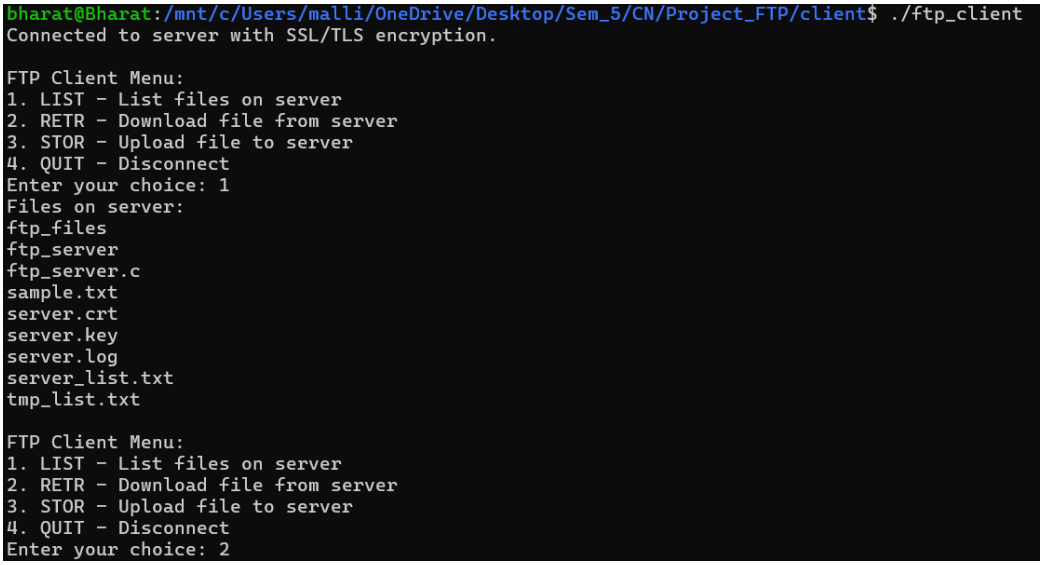

Figure 2: Client output showing the list of files retrieved from the server.

8 Conclusion

In conclusion, this project successfully developed a secure FTP client-server application utilizing SSL/TLS encryption to ensure the protection of file

transfers. By addressing the buffer overflow warnings and refining the process of formatting FTP commands, the project achieved a reliable and secure file transfer system. The lessons learned during the development process, particularly regarding buffer management and security, highlight the importance of robust string handling and encryption protocols in secure networked applications.