CS176B: Network Computing

FTPxpress

Project Deliverable C

Ben Patient
Danish Vaid
Jake Can

March 18, 2018

Contents

Abstract

Our team wanted to explore FTP and get more familiar with it, so we chose this project with the hope of learning this protocol and maybe making some improvements since its initial release in 1971. We chose to focus on making FTP more secure because inherently FTP is unencrypted and leaks all information. Though SFTP does exist, it is FTP running over a SSH tunnel so it does not modify the native FTP protocol - we wanted to introduce security natively and were able to accomplish that while retaining linear transfer rates.

Team Members:

- Ben Patient
- Danish Vaid
- Jake Can

Team Size Justification: Having been in the same capstone team for CS 189A/B the three of us established a good workflow and our similar schedules let us coordinate and work well together. We hope to use these advantages to build a solid project and have many ideas for features we could implement and networking challenges for us to overcome.

Introduction

FTP was designed when the internet was first becoming the internet and was not designed with modern issues in mind. FTP has many flaws in management and security. All of the current solutions to FTP's problems are protocols built on top of FTP but they are just treating symptoms and not the cause. The root cause is FTP, a 35 year old protocol in need of reform. We would like to implement a file transfer protocol that is secure and customizable to how the user would like to handle his or her file transfers. Our implementation is light weight and manages more back and forth between the user and the program to ensure the user's preferences are heard.

Vanilla FTP Research

FTP stands for File Transfer Protocol. FTP was first introduced in 1971 and one of the oldest networking protocols that is still used today. Its original purpose was to make transferring files from one host to another easier. To start a FTP session the client creates a TCP connection with the server and binds its port to a port on the server. Data can then be send between client and server until the session is ended. This data is unencrypted clear text that can be read by anyone who captures these packets. There are other protocols that make FTP secure like SFTP or FTPS but FTP itself is not inherently secure.

One of FTP's biggest flaws is security. Data sent between clients and servers is unencrypted. This makes FTP susceptible to packet sniffing and man in the middle attacks. This is an issue from the second an FTP session is started. To begin a FTP session the client must provide a username and password and this information could be easily intercepted by a hacker who would then gain access to the server. Additionally, weak passwords can expose the server to brute force attacks in which the hacker guesses the weak password and gains access. The openness of FTP also makes it possible for hackers to exploit with standard FTP commands. It is possible to create a denial of service attack by

creating FTP client connections on all available ports of the server causing its resources to be fully utilized. FTP inherently does not come with a filtering scheme for incoming connections. Most secure implementations of FTP today must be run behind tight firewalls to prevent this.

FTP also has no viable means of controlling what clients get priority for critical file transfers. The way FTP servers accept connections is on a first come first serve basis. While this is one of its strengths because it becomes a decentralized protocol, it is also on of its flaws because the server loses control of managing its clients. It also is what makes it vulnerable to denial of service attacks.

Project Overview

For our project, we are implementing our own version of a FTP server, client, and protocol. In doing this we are also implementing custom headers to tailor the protocol to our priorities and for the features we want to implement. These headers would include information on file, packet ordering, payload size, etc. Looking more into these headers, we decided that it would be better to have a custom handshake that exchanged meta-data, and other such data explained below. Upon the basics of FTP, we added network security features (for user privacy and ensuring data integrity).

Project Goal Summary

We are delivering a basic FTP protocol with our custom headers and a minimal terminal UI for usage. The system supports basic commands such as list files, change directory, print directory, download, and upload. Upon that, our protocol features end-to-end encryption, using RSA public/private key pairs to exchange a symmetric key during the initial connection set up and then using that symmetric key for ECB encryption. Our physical deliverable is our source code, and this report that describes the project, its completion and usage, and protocol analysis.

Custom Initialization Handshake

Our handshake involves two main agreements. We make the assumption that prior to connection, the client has the server's public key in an asymmetric key pair encryption scheme. The client uses this public key to encrypt the initialization packet which is followed by the server decrypting with it's own private key. This ensures the client that it knows that it is communicating with the server. The second piece of information that is agreed upon is the client ID. This is established by the server and then accepted by the client. The initialization packet also serves to have the server and the client agree on a symmetric key that will be use for ECB encryption for all of the following packets.

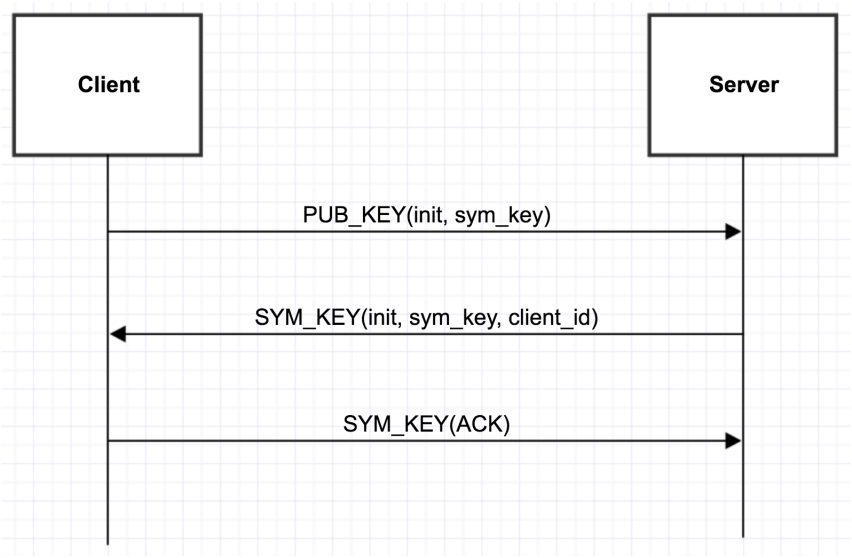

Figure 1: Initialization Handshake

Custom File Handshake

This process will set metadata for each file transfer. The benefit of including a handshake is to reduce the number of headers, therefore reducing overhead on each packet sent so there is more data per packet. Our metadata would include file name, file type, FUID (file unique identifier), and client ID. The client ID field will pair the file with the client. Knowing this ensures which file is being transferred by who, and can also be used to prevent collisions of file names.

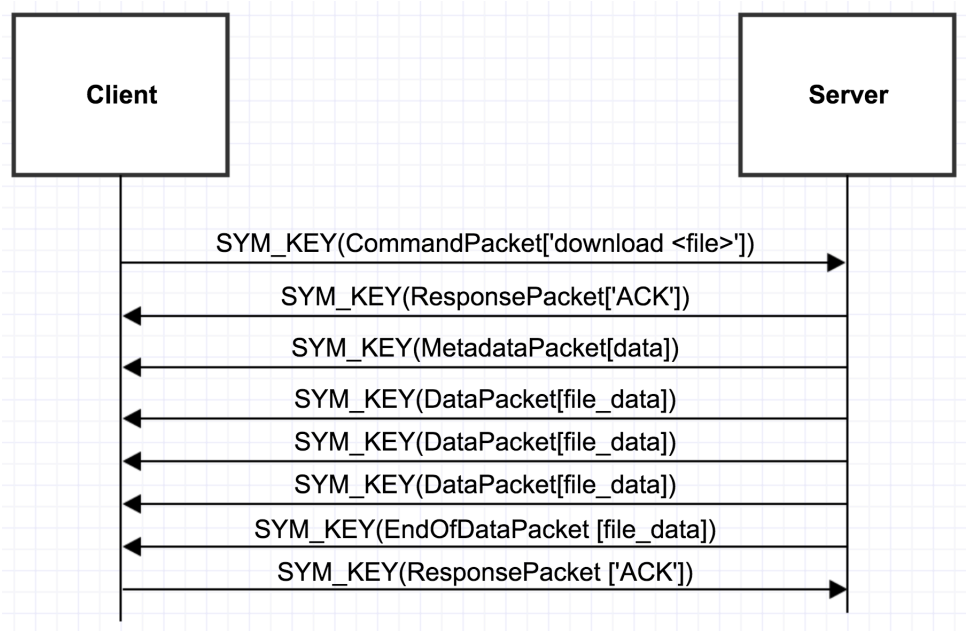

Figure 2: Example File Download

Custom Headers

Note: Many of our custom headers became a part of the custom handshake to improve performance and efficiency.

Sequence Number

This is will be a 4 hex-decimal character describing the sequence number of this packet. This is used to rebuild the packets of the file in the correct order of data of the original file.

File Unique Identifier (FUID)

This is will be a 4 digit number assigned to each file transfer. The server will assign an ID to each file that is being uploaded or downloaded. This ID will pair with the file name and type so the header will only include the ID, and not the file name and type, thus reducing overhead.

Packet Type

This is will be a 1 character information that tells whether the packet contains command data, meta-data, or file data. We can use this character flag to know what logic to run on the packet payload.

Custom Packets

CommandPacket

Command Packets contain data that sends information to the server about the command being executed, out of: ls, cd, pwd, download, and upload.

MetaDataPacket

Meta Data Packets are composed of file_uid, file_name, file_type, and client_id. This packet is sent to the server/client to give the other file context that is to be transmitted.

DataPacket

Data Packets are composed of file_uid, seq_num and data. The file_uid and seq_num are used a unique tuple to ensure the packet is for the current file being transmitted and the metadata about the file. The data simple contains the file data in this packet.

EndOfDataPacket

End Of Data Packets are used to signify to the server/client that the current file transfer has finished.

ResponsePacket

Response Packet is used by the client and server to send 'ACK's and used by the sever to send responses to the client's requested commands.

InitializerPacket

The purpose and use of the Initializer Packet has been explained above in the section `Custom`

`Initialization Handshake`

Code Classes Breakdown

A breakdown of our code classes can be found in the `README.md`

Protocol Analysis

We wanted to analyze our protocol and see how it stacks up against real use, unfortunately we could not compare against popular FTP software such as CyberDuck, FileZilla, etc because we do not have a server to test with and so our locally testing protocol against remotely tested software would not be a fair comparison.

Also, it is important to note that in an attempt to simulate some network delay we put a `sleep(0.01)` that puts a 10 millisecond delay for each packet.

Speed Test with Increasing File Sizes

File Size	Upload	Download
Mb	Sec.	
0.01	0.00749	0.0133
0.1	0.0468	0.0495
1	0.3824	0.4207
10	4.067	3.9934
100	40.9116	40.8030
500	204.3581	203.8860
1000	405.6799	409.6480

Table 1: Raw Data

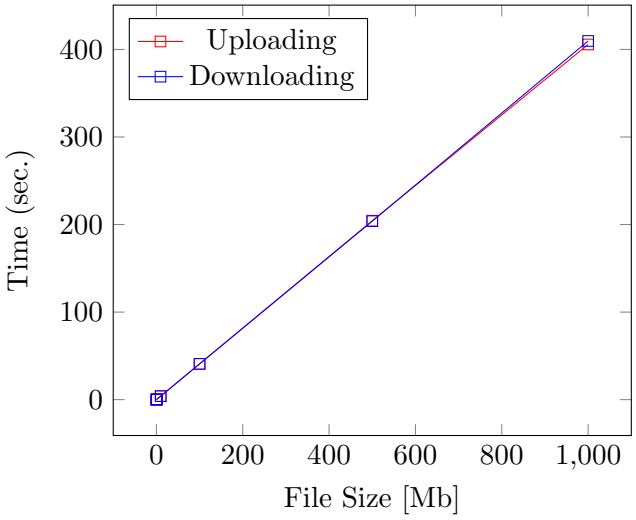

Table 2: File size vs Transmit Time

This data shows that the increase in transfer time is linear with file size growth. This is good as, if the slope was higher than linear then the transfer time would grow faster than the file size, lower than the line is unrealistic without using compression or parallelization.

Timeline

Week 3: Design our headers and protocol schematics
Week 4: Successfully send over a file from 1 user to another
Week 5: Implement basic terminal UI and command support
Week 6: Implement parallelized upload/download
Week 7: Implement network security

Week 8: Run performance testing methods
Week 9: Refine code and finalize project/report.

We were able to accomplish all of our goals except for being able to implement parallelized upload/download and the difficulties are explained in 'Challenges' below.

Results

We were able to successfully create our version of a secure FTP client and server with a functioning terminal UI. The custom headers and handshake methods that we implemented proved to be efficient by reducing overhead. Our transfer times for uploading and downloading linearly increase with file size which shows our protocol is very scalable. Furthermore we were able to make it an explicitly secure protocol by initializing an symmetric encryption scheme in our RSA public/privately secured initial handshake method and encrypting all traffic sent during the session.

Demo

We are planning our demo to first sample commands such as ls, cd, and pwd; and second, demo file transfers and show the protocol process packet by the packet in the console output.

Challenges

The biggest challenge we have had was implementing multi-threading/client. We had many ideas and believed it would have been very nice to support this functionality but ran into too many issues and ultimately decided against implementing it. The biggest issue was reading in the data stream from the socket. We believed it would be cleaner to have one central thread to accept all messages and pass these messages to the respective ConnectionHandler thread, which would process the message and either update it's state or respond to the client. Unfortunately, we were not able to send data through threads simply. We had some ideas but they were either very inefficient or poor coding practices and would take too much time to implement. Another issue with this challenge was that we would decrypt the message before sending it to the respective ConnectionHandler because we needed to know which handler should handle it by checking the client ID field within the packet. Unfortunately, to decrypt, we would need the key of the respective connection, but we wouldn't know which key to use as we didn't know the respective connection, creating a cyclic dependency.

Another challenge we had was packetization. In order to make sure we need overfill the packet, we had to calculate the overhead of the headers in order to know how much data can be fit into the packet for each file transferred. Because we had various types of packets, there were multiple sizes of overhead. We decided all of this before we implemented encryption which lead to a confusing error once we implemented encryption. After a while, we identified that we needed to increase our overhead value as we believed the encryption scheme had some overhead in it of itself.

A high level and general, but small, challenge we had was designing the server/client. We wanted a very flexible structure in preparation for multi-threaded connections. Unfortunately,

this design required a very high investment with possibilities of abundant bugs. We then decided to assume all packets sent were received sequentially. This restricted the possibilities of what can go wrong and allowed us to focus on more important aspects of our project.

References

1. https://tools.ietf.org/html/rfc412
2. https://bridgesgi.com/wp-content/uploads/2013/10/Bridge-Solutions-Three-Significant-Risks-of-FTP.pdf
3. https://thehackernews.com/2013/12/security-risks-of-ftp-and-benefits-of.html