CS 447 : Networks and Data Communications

Programming Assignment #02

Total Points: 150

Assigned Date : Thursday, May 31, 2018
Due Date : Tuesday, June 12, 2018 @ 10:59:59 a.m.

Overview

For your second project, you will improve your email application by adding basic user authentication functionality to it. Here's the back story.

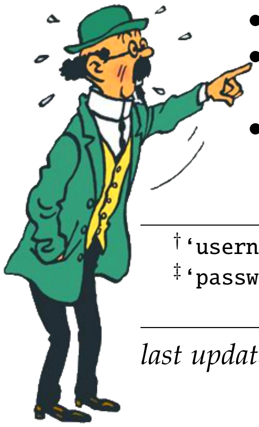
Back Story

Bizarre things are starting to happen in Calculus's email system. Haddock, for one, is quite confused about the flurry of replies he is getting to emails he didn't even send. *"What on ten thousand thundering typhoons is going on here?"* Haddock cries out furiously. *"I bet that evil Rastapopoulos is behind all this"*. Sure enough, Calculus identifies that since his vanilla SMTP connection from `client` → `server` is unauthenticated, the server is not able to validate who is sending emails. Moreover, he also discovers that anyone can login to read emails without being authenticated as well. To improve the security of his email system, Calculus plans to provide authentication capabilities at both sending and retrieving ends, and to make the server keep a log of every interaction it has with any client for postmortem diagnosis purposes.

Technical Requirements

In addition to the technical requirements from PR01, you are required to meet the following additional technical requirements.

1. Sender Authentication: Provide support for the SMTP command **`AUTH`**. Read Section 4 of the SMTP RFC found at `https://tools.ietf.org/html/rfc4954` for a description of the command.



   - The **`AUTH`** command is issued between the **`HELO`** and **`MAIL`** commands in the SMTP sequence.
   - Following the **`AUTH`** command, the server replies back with code **`334 dXNlcm5hbWU6`**[†] prompting the user to enter his/her username. (e.g. `username@447.edu`).
   - Once the user sends his username, the server then responds back with code **`334 cGFzc3dvcmQ6`**[‡] prompting the user to enter his/her password.

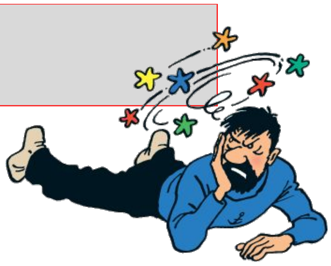
[†] `'username' in base64`
[‡] `'password' in base64`

   - A successful (or failed) authentication is marked by the appropriate reply code. Read Section 6 of RFC #4954 for status codes and implement the appropriate ones.

2. To keep the project within the time spec, the following simplified password strategy is proposed.
   (a) The first time the user responds to **334 dXNlcm5hbWU6**, the server replies back with a 5-digit randomly generated password over reply code **330** (instead of code **334 cGFzc3dvcmQ6**). The server adds 447 to this number, encodes it in base64, and stores the encoded password along with the corresponding username in a hidden master password file named ".user_pass" inside the **db** folder.
   (b) Upon receiving the 330 code and the temporary password, the client immediately terminates the current connection, waits for 5 seconds, and re-initialize a fresh connection.
   (c) On all successive connections, the client responds to server's request for password, i.e. **334 cGFzc3dvcmQ6**, by typing in the previously received password.
   (d) The server adds 447 to the received password, encodes it in base64, and checks the encoded value against the stored value in the password file. If these two values match, the authentication phase is successful.
   (e) Your client programs will take the plain text `username` and `password` entered by the user and encode it in base64 before sending. In otherwords, the server receives base64 encoded `username` and `password`.
3. Only when the authentication is successful the user is able to proceed to the next command in the SMTP sequence, i.e., **MAIL FROM** command.
4. Receiver Authentication: This is similar in function to the sender authentication. Due to time constraints, we'll use a simplified HTTP authentication mechanism instead of following HTTPS standards.
   - Upon `receiver` → `server` connection, use the same procedure as item 2 above for receiver authentication. Make your server prompt for username and password, compare the values against the stored value in the password file. You will have to go through a registration phase just like in receiver authentication. Use the same reply codes.
5. Only successfully authenticated users are allowed to download unread emails.
6. Server Incident Management: The server(s) keep an active log file (named `.server_log`) of the connection activities. Every time the server sends or receives a message, a log entry is added to this log file. More specifically, each log entry (single line) has the following format:

Answer

```
timestamp from-ip to-ip protocol-command message-code description
```



Logistics

All PR01 Logistics requirements applies to PR02. The following additional requirements are listed mostly for clarification purposes.

- You should make sure that only the user authorized is allowed to send emails. What this means is basically you should check at the client end if the username that was entered during the login process matches that with the **mail from** field.
- Your sending client is not supposed to autofill the `@447.edu` suffix of the email address for the user.

- All SMTP command validations, including the infamous end of message dot (.), and error code generation should be done at the server side and not at the client side. For all practical purposes, the sending client is just a message relay.
- For the receiver, when the user successfully logs in, he/she should be greeted with the unread number of messages in the server. The user should be given the option to either selectively read a message or read all messages.
- The receiver is not expected to type the HTTP GET message syntax. Your program will do that for the user based on his preferences. Think about it. Have you ever typed in a the HTTP GET syntax on your browser before visiting a web page?
- Emails read by the receiver should be downloaded to his (mock) local machine. You are allowed either keep the original copy in the server (IMAP standard) or to delete it (POP3 standard) after it's downloaded. However, on a successive login the number of unread message count should be updated accordingly.
- You are programming two different servers (SMTP over UDP and HTTP over TCP), which could technically be running on two different end systems. Think about this when you design/redesign your server code.

Instructions

- **Start early!!**
- **Take backups of your code often!!**.
- Follow a good coding standard. Use a Google style guide appropriate for your favorite programming language found here `https://google.github.io/styleguide/`, if you don't already follow one.
- The due date of this assignment is **Tuesday, June 12, 2018 @ 10:59:59 a.m.** A dropbox will be opened for submission through Moodle. This is a **firm deadline** as the final assignment does not include the typical 48-hour late penalty period.

Deliverables

A complete solution comprises of:

- A short report (max 5 pages) in **PDF** format of the design and implementation of your system. Your report should include the followings:
  - Introduction
  - Design choices and protocol/reply codes used.
  - The output of a sample run (including screenshots where applicable).
  - Summary and Issues encountered (if applicable).
- A short **`readme`** file with compilation instructions. A **`makefile`** is mandatory if your solution involves running multiple compilation instructions. The only exception, most likely, would be Python.
- A compressed tarball of the directory containing your source code. **Do not** include executables, folders created by your programs, or your test emails in this tarball. To create a compressed tarball of the directory source, use the following command: `tar -zcvf siue-id-pr2.tar.gz source/`.
  e.g. `tar -zcvf tgamage-pr2.tar.gz PR02/`.

Caution: File formatting standards (PDF, `readme`, `make`, and `tar.gz`) as well as some of the logistics requirements are set forth to streamline the grading process. Submissions that take a unnecessarily long time grade due to not following the standards listed in this document will be subject to penalties.

Collaborating on ideas or answering questions is always encouraged. Most times, I find that you learn a lot from your peers. However, do not share/copy/duplicate code from others. If you use code found online, remember to cite their source in your report. Issues related to academic integrity and plagiarism have **ZERO** tolerance.

Useful Resources

- Linux Man pages – found in all linux distributions
- Beej's Guide to Network Programming – A pretty thorough free online tutorial on basic network programming `http://beej.us/guide/bgnet/output/print/bgnet_USLetter.pdf`
- Simple Mail Transfer Protocol RFC #2821 `https://tools.ietf.org/html/rfc2821`
- Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content RFC #7231 `https://tools.ietf.org/html/rfc7231`
- Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing RFC #7230 `https://tools.ietf.org/html/rfc7230`
- SMTP Service Extension for Authentication RFC #4954 `https://tools.ietf.org/html/rfc4954`
- Base64 encoding/decoding:
  - C/C++
    - glib – `https://developer.gnome.org/glib/stable/glib-Base64-Encoding.html`
    - openssl `http://fm4dd.com/openssl/manual-crypto/BIO_f_base64.htm`
    - GNU coreutils `http://www.gnu.org/software/coreutils/coreutils.html`
  - Java 8 – `https://docs.oracle.com/javase/8/docs/api/java/util/Base64.html`
  - Python 2 – `https://docs.python.org/2/library/base64.html`
  - Python 3 – `https://docs.python.org/3.5/library/base64.html`
  - Additional Resource – `http://rosettacode.org/wiki/Base64_encode_data#C`

If you use any other resources, make sure to cite those in your report. Using online resources does not mean you are allowed to copy and use someone else's code for your purpose. Such incidents, if detected, will be treated as academic dishonesty.