CS447 : Networks and Data Communications
Programming Assignment #03
Total Points: 150

Assigned Date   : Wednesday, November 09, 2022
Due Date   : Wednesday, November 30, 2022 @ 01:29:59 p.m.(hard deadline)

Overview

For your final project, your objective is to improve the overall functionality of your email application by adding
user authentication and authorization (on client-side), and *p2p-like* server-to-server communication (on server-side).
NOTE: This assignment as a hard deadline with no grace period. Here's the back story.

Back Story

Soon after deployment, bizarre things started happening 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 on the
receiving (HTTP) side without being authenticated.
To improve the security of his email system, Calculus plans to add authentication functionality through a simple
username-password combo, and make servers audit client activities through event logs; the servers will now log
every interaction it have with other hosts for postmortem diagnosis purposes. Calculus also plans support sending
emails to domains other than 447f22.edu domain.

Technical Requirements

All PR02 technical requirements apply to PR03. The following requirements must be met **in addition**.

1. Sender Authentication: Provide support for the SMTP command **AUTH**. Read Section 4 of the SMTP Authenti-
   cation Extension 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 in username@447f22.edu).
   - Once the user sends his username, the server then responds back with code **334 cGFzc3dvcmQ6**[‡] prompt-
     ing the user to enter his/her password.
   - 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.

   [†]'username' in base64
   [‡]'password' in base64
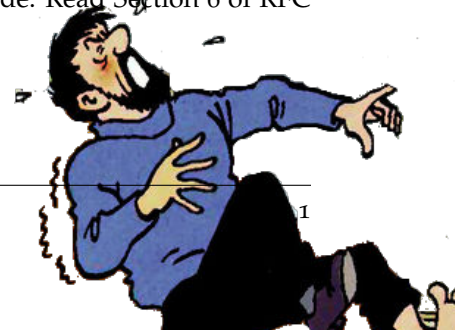
2. To keep the project within the 2-week time spec, the following simplified password strategy is proposed.

   (a) All passwords are generated at the server. Password change mechanisms are outside the scope of this project.

   (b) The first time the user responds to **334 dXNlcm5hbWU6**, the server will:
      - Generate a 5-characters long random alpha-numeral string;
      - Encode the generated password in base64 and send it to the client over reply code **330** (instead of code **334 cGFzc3dvcmQ6**);
      - Prefix the random alpha-numeral password string with SNOWY22 (for salting), encodes it in base64, and store the encoded password along with the corresponding username in a <u>hidden</u> master password file named **".user_pass"**. This master password file should reside inside the **db** folder (and should not be shared with any clients, users, or other servers).

   (c) Upon receiving the **330** code and the encoded password, the client should:
      - Decode and display/print the plaintext password on the standard output;
      - Warn the user that the connection will *"reset"*, and enter a 5 second countdown;
      - At the end of the countdown, disconnect the current connection, and automatically reconnect to the server (from a new socket address). This will force the user to enter the proper authentication process (enter both the correct `username` and `password`).

   (d) On all successive connections, the client responds to server's request for the password, i.e. **334 cGFzc3dvcmQ6**, by typing in the previously received alpha-numeral.

   (e) The user is allowed to enter usernames and passwords in plaintext on the terminal. However, your `client` and `server` programs should not exchange paintext `password`s. In otherwords, the server should only be receiving encoded strings as `password`s.

   (f) The server validates the user's authenticity by decoding and salting the incoming string, encoding the resulting string in base64, and comparing it with what's stored in `.user_pass` against the corresponding `username`. If the two values match, the authentication phase is successful. **Note**: The password file is indexed by `username`.

   (g) Note that the `username` used for authentication does NOT necessarily need to match the sender moniker (i.e., <sender>@447f22.edu) in the sender's email. Such an attempt will fail the authorization step listed below under Logistics.

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. <u>Receiver Authentication</u>: We will forego receiver authentication and password generation in the interest of time.
   - If there is an inbox at the server for the `username` entered, print the number of unread emails the server is holding for that user. This unread email count should be appropriately updated on successive logins.
   - If there is no inbox for the user or if the `username` is invalid, print an error response and disconnect.

5. <u>Server Incident Management</u>: 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 reply-code description`

6. <u>Inter-Domain SMTP</u>: Your email application is now required to support communications between the 447f22.edu domain (*from PR02*) and other domains. For example, `tintin@447f22.edu` can now email `haddock@454f22.edu` (similar to how someone can email from `gmail` to `yahoo`).
   - The sender's SMTP server needs to directly communicate with the receiver's SMTP server to transfer the email from source to destination. This behavior is akin to how real-world SMTP servers work (*see fig. 2.16 in the textbook*).

- Note that during this `server` → `server` exchange, the sender's SMTP server "*behaves*" like a SMTP client relative to the receiver's SMTP server. As such, this exchange must follow the `sender` → `server` interaction specification from PR02. The process, however, is automated without any user intervention.
- The server incident management (item 6 above) still applies to even `server` → `server` communication.
- Each server knows only about remote domains explicitly listed in their server configuration files. Here's an example of an augmented server configuration file skeleton.

```
server.conf
SMTP_PORT=
HTTP_PORT=

[REMOTE_DOMAIN] //e.g.  454f22
IP=
PORT=
```

The `[REMOTE_DOMAIN]` block can be duplicated to support any number of different domains. We will at least test with 3 (or more) different domains. You may assume all domains are under the `.edu` top-level domain.

- Servers may authenticate each other using the magic word **`rasta#22smtp`** sent as an argument with the **`AUTH`** command. The receiving server will user this magic word to authorize the incoming connection as another peer server. This will allow the receiving server to by-pass the authentication step and password checking among smtp peers.

Logistics

All PR02 Logistics requirements applies to PR03. The following additional requirements are listed for clarification purposes.

- Only <u>authenticated</u> and <u>authorized</u> senders are allowed to use particular server to send emails. Authentication is described above in technical requirements. Here, authorization means two things:
  - The server must check the `username` entered during authentication matches the sender moniker of the **`mail from`** field.
  - The server must prevent users from sending emails from alien domains. For example, `Tintin` should **NOT** be able to use the SMTP server for the `447f22.edu` domain to send emails as `tintin@454f22.edu`
- 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 improved user friendliness, prompt the user for appropriate command parameters (instead of asking for both command syntax and parameters) during the SMTP interaction.
- Emails read by the receiver should be downloaded to their (mock) local machine. You are allowed to either keep the original copy in the server (IMAP standard) or to delete it (POP3 standard) after it's downloaded. However, on a successive logins, the number of unread message count should be updated accordingly.
- In addition to the audit log, all servers should print the log entry of all activities real-time (i.e. as they happen) on the standard output.

Extra Credit

Extra credit is only considered when core requirements are met.

- <u>Receiver Authentication</u>: 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 for new receivers just like in the sender registration listed under 2.(a) above. You may also use the same reply codes.

- Only successfully authenticated users are allowed to download unread emails.

Instructions

- **Start early!!**. This is a fairly loaded assignment.
- **Take backups of your code often!!**.
- Follow a good coding standard. Use one of Google's coding standard found here `https://google.github.io/styleguide/`, if you don't already follow one.
- Your code must compile and run on a typical Linux setup. Neither the instructor nor his graders will use (or entertain the use of) any IDE to test your implementation. Be sure to test command line compilation and run before submission.
- Implementation language must be C/C++.
- **Absolutely do not** include executables, folders created by your programs, hidden files, version control repositories, or any irrelevant files in this tarball. All project relevant file formatting standards (**PDF, README, .txt, .tar.gz**) will be strictly monitored and are subject to penalties.
- The due date of this assignment is <mark>**Wednesday, November 30, 2022 @ 01:29:59 p.m.(hard deadline)**</mark>. This is the final programming assignment for this semester, hence there is no grace period. A Moodle dropbox will be opened for submission.
- This assignment can be fully developed using the socket API of your programming language and basic I/O API. Use of other packages/libraries without the instructor's permission is not permitted.

Deliverables

A complete solution comprises of:

- A short report of the design and implementation of your system. The report should be **PDF** format. At a minimum, your report should include the following sections:
  - Introduction: Your objective and what you hope to gain from the assignment.
  - Overall design, specific design choices, and reply codes used.
  - The output of a sample run. Include plenty screenshots wherever applicable. In situations were we can't verify expected behavior, your screenshots maybe considered for partial credit.
  - Summary and Issues encountered. What you were able to achieve from your own objectives (from the introduction) as well as project specifications. Make sure to explicitly list functionality you failed to implement (or buggy).
- A compressed tarball that contains:
  - a directory containing (only) your source code and config files. **Do not** include executables, folders created by your programs, or any other files not specifically listed here as required.
  - A short `README` file with compilation and run instructions.
  - A `makefile` (**mandatory**) to compile your code especially if it involves compiling multiple executables with flag options.

> To create a compressed tarball of the directory source, use the following command:
> `tar -zcvf siue-id-pr3.tar.gz source/`.
> e.g. `tar -zcvf tgamage-pr3.tar.gz PR03/`

Collaborating on ideas or answering each other's 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, including online sources. If you find a useful article or a source code online, firstly cite it in your report, and secondly, get the idea from it and do your own implementation. Absolutely do not directly copy-paste code from online sources.

The exercise is meant for you to learn network programming, not to test your googling abilities. The instructor actively uses MOSS http://theory.stanford.edu/~aiken/moss/ to check for software similarity. Issues related to academic integrity and plagiarism have **<u>ZERO</u>** tolerance and will result in a failing grade for the course.

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 for C/C++ https://beej.us/guide/bgnet/
- Linux Socket Programming In C++ – https://tldp.org/LDP/LG/issue74/tougher.html
- The Linux HOWTO Page on Socket Programming – https://www.linuxhowtos.org/C_C++/socket.htm
- 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++
        * openssl http://fm4dd.com/openssl/manual-crypto/BIO_f_base64.htm
        * glib – https://developer.gnome.org/glib/stable/glib-Base64-Encoding.html
        * GNU coreutils http://www.gnu.org/software/coreutils/coreutils.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.