CS447 : Networks and Data Communications *(sec. 02)*

Programming Assignment #02

Total Points: 150

Assigned Date : Wednesday, March 17, 2021
Due Date : Wednesday, March 31, 2021 @ 01:29:59 p.m.

Overview

For your second project, your objective is to improve the overall functionality of your email application by adding user authentication (on client-side) and *p2p-like* server-to-server communication (on server-side). 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 `447.edu` domain.

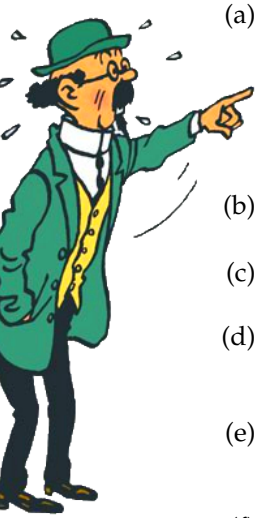
Technical Requirements

All PR01 technical requirements apply to PR02. The following requirements must be met **in addition**.

1. Sender Authentication: Provide support for the SMTP command **`AUTH`**. Read Section 4 of the SMTP Authentication 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@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.
   - 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. Here the server generates passwords for all clients. Password change mechanisms are outside the scope of this project.



   (a) The first time the user responds to **`334 dXNlcm5hbWU6`**, the server replies back with a randomly generated alpha-numeral string of length 6 over reply code **`330`** (instead of code **`334 cGFzc3dvcmQ6`**). The aforementioned alpha-numeral is the server generated *user password*. On the server-side, the server prefixes this string with `447S21` (for salting), encodes it in base64, and stores the encoded password along with the corresponding username in a hidden master password file named "`.user_pass`". This master password file should reside inside the **`db`** folder.
   (b) Upon receiving the 330 code and the server generated password, the client immediately terminates the current connection.[§] The user must re-initiate a connection to the server for proper authentication.
   (c) 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.
   (d) The server validates the user's authenticity by salting the incoming string, encoding it in base64, and comparing the result with what's stored in `.user_pass` against the corresponding username. If the two values match, the authentication phase is successful.
   (e) The user is allowed to enter usernames and passwords in plaintext on the terminal. However, your client programs should only exchange base64 encoded `username` and `password` strings with the server. In otherwords, the server receives base64 encoded `username` and `password`.
   (f) Note that the `username` used for authentication does NOT necessarily need to match the sender moniker (i.e., `<sender>@447.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. 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 for new receivers just like in the sender registration listed under 2.(a) above. You may also 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 reply-code description
   ```

7. Inter-Domain SMTP: Your email application is now required to support communications between the `447.edu` domain (*from PR01*) and other domains. For example, `tintin@447.edu` can now email `haddock@490.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 PR01. However, the process is automated without any user intervention.
   - The server incident management (item 6 above) still applies to even `server` → `server` communication.
   - Each server's own domain, that of the remote server(s), and their socket addresses are provided through the corresponding server configuration file. Here's a skeleton sample configuration file.

[§] It probably makes sense to print the decoded alpha-numeral value to the standard output before terminating this connection.

server.conf

```
[SELF_DOMAIN]
SMTP_PORT=
HTTP_PORT=

[REMOTE_DOMAIN] //remote domain #1
IP=
PORT=
```

The `[REMOTE_DOMAIN]` block can be duplicated to support any number of different domains. We will at least test with 2 (or 3) different domains. You may assume all domains are under the `.edu` top-level domain.

Logistics

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

- Only authenticated and authorized 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 for authentication matches the sender moniker of the **`mail from`** field.
  - The server must prevent users from sending emails pretending to be from other domains. As an example, Tintin should NOT be able to use the SMTP server for the `447.edu` domain to send emails as `tintin@490.edu`
- Similarly, only authenticated and authorized receivers should be allowed to retrieve emails using a particular (HTTP) server.
- 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.
- for improved user friendliness, prompt the user for appropriate command parameters (instead of asking for both command syntax and parameters) during the SMTP interaction.
- Similarly, the receiver is no longer 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.
- 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.

Instructions

- **Start early!!**. Ask questions through `Discord := Projects` channel before making costly incorrect assumptions.
- **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 grader will use (or entertain the use of) any IDE to test your implementation. Be sure to test command line compilation and run before submission.
- **Absolutely do not** include executables, folders created by your programs, hidden files, version control repositories, or your test emails in this tarball. All project relevant file formatting standards (**`PDF, README, .email, .txt, .tar.gz`**) will be strictly monitored and is subject to penalties.
- The due date of this assignment is **Wednesday, March 31, 2021 @ 01:29:59 p.m.**. A Moodle dropbox will be opened for submission.
- Based on past student experience, multi-threading is not as obvious as it may look. I suggest you to first get a single-threaded version working correctly and then think about extending it to multi-threading.
- 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 instructors permission is not permitted. The use of an appropriate base64 library is allowed. Check Useful Resources section below for some resources.

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. Python based submissions should use the `makefile` to echo the content of your `README` file.

To create a compressed tarball of the directory source-dir, use the following command:
`tar -zcvf siue-id-pr2.tar.gz source-dir/`.
e.g. `tar -zcvf tgamage-pr2.tar.gz PR02/`

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. The exercise is meant for you to learn network programming, not to test your googling abilities. 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 14 – https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Base64.html
  - Python 2 – https://docs.python.org/2.7/library/base64.html
  - Python 3 – https://docs.python.org/3/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.