Postman Impersonation

This assignment is worth **20%** of your overall grade for the course.

Due: Week 13, Thursday, November 3rd at 23:59:00, Sydney local time.

You will implement in Python some email exchange applications that supports the `SMTP-CRAM` protocol. This includes a client, a server and an eavesdropper (middle-man attacker), which can apply MitM attack to the authentication mechanism. Unix tools like `netcat` and `telnet` can help test your program.

To limit the workload, we provide a simplification on the original `SMTP` and `SMTP-CRAM` protocols. We assume all programs are supposed to be run and tested locally and they are not required to be extensible to support TCP connection via the Internet or to be used in real world. No actual data should be sent and received from the Internet, and everything should be contained in the local loopback. We also assume any syntactically valid message under the protocol can be accepted. E.g., No actual check is needed on any email or IP address, we accept any syntactically valid messages including addresses.

The communication between the email exchange applications has to follow a popular protocol for email exchanging, the Simple Mail Transfer Protocol (`SMTP`) protocol with one authentication extension (`CRAM-MD5`), the SMTP Service Extension for Authentication by Challenge-Response Authentication Mechanism. The authentication type associated is called `CRAM-MD5` and the full protocol is named `SMTP-CRAM` .

However, `CRAM-MD5` only allows the server to verify the client and doesn't provide any server authentication, therefore its usage is limited and less preferable than other stronger mechanisms. To demonstrate `CRAM-MD5` indeed has its weakness, the eavesdropper should be implemented and a MitM attack can be performed.

On the high level, the following needs to be implemented:

- All programs are capable of:
  - Log all `socket` transactions in a specific format and output to `stdout` .
- The `SMTP-CRAM` server is capable of:
  1. Prepare for any incoming client connection.
  2. Receive emails from a client and save to disk.
  3. Additionally, allow client authentication.
  4. Allow multiple clients to connect simultaneously.
  5. Termination upon receiving a `SIGINT` signal.
- The `SMTP-CRAM` client is capable of:
  1. Read mail messages on disk.
  2. Send emails to the server.
  3. Additionally, allow client authentication.
  4. Termination after finishing sending all emails.
- The `SMTP-CRAM` eavesdropper (middle-man attacker) can do active eavesdropping between a pair of given server (E.g.,the real server) and client (E.g.,the real client). It can intercept all `socket` messages passing between the real server and the real client without being discovered. You can think of the eavesdropper as a combination of one valid client and one valid server, in such way it can impersonate the real client without letting the real server to discover its eavesdropping. This means it is capable of:
  1. Prepare for being connected to by the real client and connecting to the real server.
  2. Capture the email sent by the real client and save to disk, without altering the content.
  3. Additionally, comprise any client authentication.
  4. Termination.

The assignment has four tasks that you will need to implement and thoroughly test. You will be provided a test suite to assist in developing your solutions.

The four tasks are:

1. Implement a client that supports the `SMTP-CRAM` protocol.
2. Implement a server that supports the `SMTP-CRAM` protocol.
3. Implement an eavesdropper (middle-man attacker) that secretly relays the communications between the client and the server that supports the `SMTP-CRAM` protocol.

4. Write a report in your own words on the `SMTP-CRAM` protocol.

A bonus task can be implemented if the Task 1, 2 and 3 are completed and verified by public testcases:

- Extend the server to support multiple connections from clients.

Please read the whole specification to better understand these tasks.

*Background Story

* Note: It is totally OK to skip reading this part if you just want to do your assignment. Trust me, I am 100% honest. ~~Or maybe I am not?~~

** Actually after reading this section, the whole assignment should make more sense.

You are a rookie Webrunner (hacker) that interns at a game company, ***Asaraka*** in the Sun City. (This should not be confused with ***Arasaka***.)

Your first job is to do a case study on an insecure ancient email protocol over `socket` called `SMTP-CRAM` . Your boss David told you that `SMTP-CRAM` does not provide server authentication and cannot be resistant to man-in-the-middle attack. But as a rookie, you don't know what David was talking about, and decided to do some research and work on it on your own. It is your first job, you'd better not mess it up, you think to yourself.

`SMTP-CRAM` is so ancient that you cannot even find a concrete implementation of the protocol. You decide to implement it first and that includes a `SMTP-CRAM` server and a `SMTP-CRAM` client.

After that, you want to build an actual middle man attacker server to break the protocol. Hopefully, you will impress your boss and get a raise. ***Hopefully***. Well, it's the Sun City, who knows what will happen next. Anyway, your future awaits, you'd better start now!

Glossary

`SMTP-CRAM` protocol

The Simple Mail Transfer Protocol (`SMTP`) with Extension for Authentication by Challenge-Response Authentication Mechanism. Or `SMTP-CRAM` protocol.

A communication protocol is a system of rules that allows two or more entities of a communications system to transmit information.

The `SMTP-CRAM` protocol, in the context of this assignment, is a stateful communication protocol defined in the application layer. As an application layer protocol, it relies on the commonly transport layer protocol: the Transmission Control Protocol (TCP). Note that the original `SMTP` can be implemented on the User Datagram Protocol (UDP) layer, but in this assignment UDP is out-of-scope.

Note that a large proportion of this specification is adapted based on a collection of RFC documents, they are

- RFC2821: Simple Mail Transfer Protocol,
- RFC2554: SMTP Service Extension for Authentication,
- RFC2104: HMAC: Keyed-Hashing for Message Authentication,
- RFC2195: IMAP/POP AUTHorize Extension for Simple Challenge/Response.
- RFC5234: Augmented BNF for Syntax Specifications: ABNF

We acknowledge this fact and we pay tribute to the past and the present of the Internet Engineering Task Force, the History of the Internet and all efforts towards creating these protocol standards.

However, it is strongly **NOT** recommended to directly follow the referred documents above, because the assignment is neither a subset nor a superset of the collection. As a simplification, some details in RFC have been omitted intentionally; and to adapt it into an assignment, other details have been added.

Request and Response

Data is exchanged through a sequence of request-response messages which are exchanged by a transport layer protocol connection. A client initially tries to connect to a server establishing a TCP connection. The client sends requests to the server. Upon receiving the request, the server sends back a response message.

ABNF (Augmented Backus–Naur form)

ABNF is a popular formal method of describing communication syntax. It is used in this assignment to help us formally define syntax of `SMTP-CRAM` Request and Response. More information and examples can be found on the Internet. E.g.:

- Wikipedia
- RFC5234

Note in ABFN, new lines do not mean literal new lines. One definition may be broken into multiple lines just to prettify the document.

We predefine the rules below as a convention. However, you should also visit the Wikipedia page to understand the quantifiers such as `*` .

```
;;                  ; LEADS A COMMENT
OCTET               = <any 8-bit sequence of data>
CHAR                = <any US-ASCII character (octets 0 - 127)>
UPALPHA             = <any US-ASCII uppercase letter "A".."Z">
LOALPHA             = <any US-ASCII lowercase letter "a".."z">
ALPHA               = UPALPHA | LOALPHA
DIGIT               = <any US-ASCII digit "0".."9">
CTL                 = <any US-ASCII control character (octets 0 - 31) and DEL (127)>
CR                  = <US-ASCII CR, carriage return (13)>
                    ;; \r in Python
LF                  = <US-ASCII LF, linefeed (10)>
                    ;; \n in Python
SP                  = <US-ASCII SP, space (32)>
HT                  = <US-ASCII HT, horizontal-tab (9)>
                    ;; \t in Python
<">                 = <US-ASCII double-quote mark (34)>
HEX                 = "A" | "B" | "C" | "D" | "E" | "F"
                    | "a" | "b" | "c" | "d" | "e" | "f" | DIGIT
Snum                = 1*3DIGIT
                    ;; <one, two, or three DIGIT representing a decimal
                    ;; integer value in the range 0 through 255>
IPv4-address        = Snum 3("." Snum)
                    ;; four Snum seperated by dot, E.g.,8.8.8.8
Let-dig             = ALPHA / DIGIT
Ldh-str             = *( ALPHA / DIGIT / "-" ) Let-dig
CRLF                = CR LF
                    ;; \r\n
```

MitM (man-in-the-middle) attack

In cryptography and computer security, a man-in-the-middle attack is a cyberattack where the attacker secretly relays and possibly alters the communications between two parties who believe that they are directly communicating with each other, as the attacker has inserted themselves between the two parties.

One example of a MitM attack is active eavesdropping, in which the attacker makes independent connections with the victims and relays messages between them to make them believe they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker. The attacker must be able to intercept all relevant messages passing between the two victims and inject new ones.

This is straightforward in many circumstances; for example, an attacker within the reception range of an unencrypted Wi-Fi access point could insert themselves as a man-in-the-middle. As it aims to circumvent mutual authentication, a MitM attack can succeed only when the attacker impersonates each endpoint sufficiently well to satisfy their expectations.

Most cryptographic protocols include some form of endpoint authentication specifically to prevent MitM attacks. For example, `TLS` can authenticate one or both parties using a mutually trusted certificate authority. However, `SMTP-CRAM` is a limited protocol that cannot really prevent MitM attacks.

In Task 3, you are asked to implement an eavesdropper program that can perform active eavesdropping attack under the `SMTP-CRAM` protocol. In other words, if the attacker program is provided enough information on the victims (a

`SMTP-CRAM` client and a `SMTP-CRAM` server), it can then make independent connections with the victims and relays messages between them to make them believe they are talking directly to each other over a private connection, when in fact the entire conversation is controlled by the attacker. The attacker is able to intercept all relevant messages passing between the two victims and inject new ones.

For testing, we enforce the real client to connect to the attacker, and also the real server to accept the attacker as a client in good faith.

Configuration file

The configuration file is a simple text file where each property is assigned to a value. The `SMTP-CRAM` programs will need their own configurations to be launched.

Email transaction text file

This file is a simple text file follows a transaction text formation. The client should be able to read this type of file, and the server and the eavesdropper can parse the received `SMTP-CRAM` request then save emails on disk in this type of file.

Buffers and states

Some communication sessions are stateful, with both parties (the client and the server) carefully maintaining a common view of the current state. In general, the state of the communication can be modelled by a virtual "buffer" and a "state table" on the server. A "buffer" is a block of data associated with a "state". A "state" is a discrete step in a computation depending on some input. Any time a client sends a request or a server sends a response, the communication's state is changed. The "buffer" and "state" may be updated by the client, for example, "insert something to buffer", "clear the buffer", "move to another state", "stay with the same state", "reset the state". The client may cause the information in the buffer to be updated or discarded and the state to be returned to some other state.

In the context of sending and receiving emails, it is natural to define the following buffers:

- Source buffer, which stores the sender.
- Destination buffer, which stores the (one or more) recipient(s).
- Data buffer, which store the content of email.

The meaningful states in this assignment can be found in later section.

Personal Credentials

In this assignment, every student is assigned a pair of **confidential**, unique and random identifier and secret. You can find them in the Ed slide Personal Credentials. Specifically, you should click "Mark" to view the return messages, and they should be of the form:

```
Your Personal ID is:
ABCDEF
Your Personal Secret is:
.............................
```

This should be used when you build the client and server program, because the `AUTH` command will need them as the user name and the shared secret.

Note that you shall ALWAYS use your own personal credentials and NEVER share your credentials with other students. An impersonated use of valid credentials must indicate an exchange of confidential information or the credential generation or exchanging process is compromised. Staff will investigate into any impersonation incidents, and if the exchange of credentials between individuals is confirmed, it will result in VERY harsh deduction on your assignment.

This will be treated very seriously. In any confirmed breach incident, the individual who intentionally shares the credentials and who uses the credentials of others will face the same level of penalty. Please make sure you do not let others know your the credentials easily.

The same level of penalty will apply to those motivated individuals who attempt to crack the generating process and finally achieve their goal. Any attack in good faith is also forbidden. It is suggested that you shouldn't attempt to do so.

In short, there is no gain in sharing your secret, so please keep it secret. Thus, there is no gain in knowing others' secrets, so please don't try.

Convention

In examples, `C:` and `S:` indicate lines sent by the client and server respectively. In addition, `AC:` and `AS:` indicate lines sent by the eavesdropper (attacker) to the client and server respectively.

File formations

Configuration files

As with most web servers, the configuration files need to have porting information. Specifically:

- The server configuration should have a `server_port`, and it awaits incoming TCP registration connections at the `localhost` on the `server_addr`. It also has a `inbox_path`, a directory path to save any emails received.
- The client configuration should have a `server_port`. It sends outgoing TCP registration connections to the `localhost` at the `server_port`. It also has a `send_path`, a directory path to read any emails to be sent.
- The eavesdropper configuration should have a `server_port` and a (different) `client_port`. It relays information between the server and the client. Note, the client will be launched by having its `server_port` being the `client_port` of the eavesdropper. It also has a `spy_path`, a directory path to save any emails captured.

The configuration file is a simple text file where each property is assigned to a value. When reading the configuration file, you will need to retrieve each of needed properties and their value from the file.

- `server_port` - integer, the port the server listens on. Greater than 1024.
- `client_port` - integer, the port the server listens on. Greater than 1024. If present, should be different from `server_port`.
- `*_path` - string, a valid and writable path on disk. If multiple paths present, should be different from each other.

Configuration file example

```
server_port=1025
client_port=1026
inbox_path=~/inbox
send_path=~/send
spy_path=~/spy
```

Email transaction text file

It is encoded by ASCII in a human-readable manner, it has to have the following fields and in the exact order shown:

- Sender information.
  - One line and non-empty.
  - Email address in `<>` bracket.
  - `From: <someone@domain.com>`
- Recipent(s) information.
  - One line and non-empty.
  - Email address in `<>` bracket.
  - Separated by `,` if there are multiple address supplied.
  - `To: <other@domain.com>`
  - `To: <other@domain.com>,<other.other@domain.com>`
- Sending time.
  - One line and non-empty.
  - Date and time in RFC 5322 format.
  - `Mon, 14 Aug 2006 02:34:56 -0600`
- Subject.
  - One line and non-empty.
- Body.
  - Multiple lines in ASCII.

Note that to to ensure text files are encoded in ASCII, one could use a primitive text editor with input methods disabled. E.g., CJK language characters are not ASCII.

Email transaction text file example

```
From: <bob@bob.org>
To: <alice@example.com>,<me@carol.me>
Date: Mon, 14 Sep 1987 23:07:00 +1000
Subject: Frist Electronic Mail from Bob to Alice and Carol
Across the Exosphere we can reach every corner on the Moon.
P.S. THIS IS NOT A SPAM
```

Program behaviours

All programs

Logging

Log all `socket` transactions in a specific format and output to `stdout`. Each program will log the `socket` data **chronologically**, and the identity of the apparent sender. For multiline messages, every `CRLF` implies a line break in the log. Note, the eavesdropper always knows who the client and server are, however both the client and server are unaware if they are connected to an eavesdropper or not.

Logging example

Suppose the client connects to the server, then first sends `abc` to the server and the server replies `cba` after receiving `abc`. Both the client and the server should output the following in `stdout`.

```
C: abc
S: cba
```

Suppose the client connects to the eavesdropper (attacker), which it regards as the server, then first sends `abc` to the eavesdropper. The eavesdropper connects to the server and relays (sends) `abc` to the server. After receiving `abc`, the server replies `cba` to the eavesdropper, which it regards as the client. After receiving `abc`, the eavesdropper replies `cba` to the client.

Because the eavesdropper does not change the message, both the client and server write the same output to `stdout`, identical to the above example. However, the eavesdropper knows more:

```
C: abc
AS: abc
S: cba
AC: cba
```

Server

A server should:

1. Read the command line arguments. The configuration file path should be supplied. If not, terminate the program with exit code `1`. Parse the configuration file. If any needed property is missing or its value is invalid, terminate the program with exit code `2`. To be specific, if the program cannot write files to the `inbox_path` path, terminate the program with exit code `2`; if the program failed to bind to the given port, terminate the program with exit code `2`.
2. Prepare for any incoming client connection. Accept and establish a TCP connection from a client. Whenever the client disconnects unexpectedly, log `S: Connection lost` to `stdout` but do not terminate the program.
3. Receive an email from a client. Receive request data from and send response data to a client over TCP. When an email transaction is finished, parse and save the email transaction to a text file in the email transaction text format. The file should be named after the Unix timestamp of the sending time. If any field (e.g., sender, recipients) is missing or invalid, do not raise an error and keep the field empty. If the sending time is missing or invalid, the file should be named `unknown.txt`. If an email to be saved should share the same file name with another existing file on disk, overwrite the existing file.

4. Additionally, allow client authentication. When `SMTP-CRAM` is implemented (e.g., `CRAM-MD5` is included in the server response of `EHLO`), send the challenge data to the client and authenticate the client before accepting the email. If the client has been verified for authentication in this session, prefix the file name of the email, which is received after the authentication, with `auth.`.
5. End a client session when the client sends a `QUIT` request.
6. (Optionally) Allow multiple clients to connect simultaneously by multi-processing technique (explicitly by `fork`ing, not multi-threading).
7. Termination. Upon receiving a `SIGINT` signal, print `S: SIGINT received, closing` to `stdout`, and properly terminates ongoing client sessions and itself with with exit code `0`.

Multi-process server

* This section is related to the bonus task.

If the server can handle multiple connections from clients simultaneously, it needs to spawn multiple child processes to handle each client connection.

In this assignment specifically, there are a few additional requirements on implementation are needed:

1. `fork` implementation. Multi-threading implementation is not allowed. Only multi-processing implementation by `os.fork()` is allowed. Wrong implementation will result in manual deduction.
2. Instant `fork` after connection establishment. It is required that once the server `main` process (the process that is launched manually by the user) accepts a client connection, it should `fork` immediately to a server `child` process. In other words, the main process should only wait for new client connections and accept them. It will let its children handle the real communication in each server-client session.
3. Prefix with `[PID][ORDER]`. The bahaviour of logging and saving need a minor change. This helps the user to differentiate between the server `child` processes. Instead of using the simple convention `C:` and `S:`, the `child` processes should add a prefix `[PID][ORDER]`. When saving an email transaction file, also add a prefix `[PID][ORDER]` to the file name.
4. Server `child` process quits when client `QUIT`s. Unlike single process server, a server `child` terminates when the client `QUIT` and it sends a `221` reply.
5. Termination. When `main` receives a `SIGINT`, it should immediately stop accepting new clients and signal all alive `child` processes by `SIGINT` to force them to terminate. After ensuring all `child`s' terminations, `main` terminates itself. You should assume that no `child` process should be signaled by `SIGINT` externally, but it is nice to make the `main` server tolerant unexceptedly terminated `child`.

For example, if the server has received two connections from two clients, it may log something like this:

```
[110][01]S: 220 Service ready
[115][02]S: 220 Service ready
[110][01]C: EHLO 127.0.0.1
[110][01]S: 250 127.0.0.1
[115][02]C: EHLO 127.0.0.1
[110][01]S: 250 AUTH CRAM-MD5
[115][02]S: 250 127.0.0.1
[115][02]S: 250 AUTH CRAM-MD5
```

Where `[110][01]` means the server `main` process spawned the first (`[01]`) `child` process to handle a client, and the `[01]` `child` has process ID `110`. Similarily, we know `main` forked to `[02]` `child` with PID `115`. The `order` should never decrement and increment by 1 when a new connection is established. It won't decrement when a server `child` process terminates.

Recall the `fork` and `exec` practices in Week 3 lab, the order of the logs among `child` processes might not be completely determinastic, but the order has to deterministic for a specific `child` process.

Client

A client should:

1. Read the command line arguments. The configuration file path should be supplied. If not, terminate the program with exit code `1`. Parse the configuration file. If any needed property is missing or its value is invalid, terminate the program with exit code `2`. To be specific, if the program cannot read files in the `send_path` path, terminate the program with exit code `2`.
2. Prepare a mail message to be sent. Read and parse an email transaction text files in `send_path`. If there are more than one email text files, queue them in the dictionary order of the file names, and process only one at a time. Whenever an email transaction text cannot be parsed due to bad formation, log `C: <filepath>: Bad formation` to `stdout` and continue to the next file without retry.
3. Attempt a TCP connection to a server. If failed, log `C: Cannot establish connection` to `stdout` and terminate the client with exit code `3`. Whenever the server disconnects unexpectedly, log `C: Connection lost` to `stdout` and terminate the client with exit code `3`.
4. Send an email to the server. Send request data to and receive response data from and a server over TCP.
5. Additionally, allow client authentication. When `SMTP-CRAM` is implemented (e.g., `CRAM-MD5` is included in the server response of `EHLO`) and the email transaction text file absolute path has `AUTH` (case-insensitive) in its absolute path file, send `AUTH` to the server immediately after `EHLO`. Send the answer (to the challenge) data to the server and prove its authentication to the server before sending the email.
6. Termination. Send a `QUIT` request after finishing sending the email, then terminate with exit code `0`.

Eavesdropper

An eavesdropper should:

1. Read the command line arguments. The configuration file path should be supplied. If not, terminate the program with exit code `1`. Parse the configuration file. If any needed property is missing or its value is invalid, terminate the program with exit code `2`. To be specific, if the program cannot write files to the `spy_path` path, terminate the program with exit code `2`.
2. Prepare for being connected to by the real client and connecting to the real server. Accept and establish a TCP connection from the real client. Attempt a TCP connection to the real server. If failed to connect to the real server, log `AS: Cannot establish connection` to `stdout` and terminate the client with exit code `3`. Whenever the real server disconnects unexpectedly, log `AS: Connection lost` to `stdout` and terminate the client with exit code `3`. Whenever the client disconnects unexpectedly, log `AC: Connection lost` to `stdout` but do not terminate the program.
3. Capture the email sent by the real client, log the message and relay to the real server. Receive request data from and send response data to the real client and the real server over TCP. Output all `socket` transactions to `stdout`, write other internal information and error to `stdout`. When an email transaction is finished, parse and save the email transaction to a text file as if it is a server.
4. Additionally, comprise any client authentication. When `SMTP-CRAM` is implemented, relay the real server challenge to the real client, steal the valid answer to the real challenge, and relay the valid answer to the real server. In such a way it can pretend to be the real client as it steals the valid authenticated answer.
5. Termination. Send a `QUIT` request to the real server after receiving a `QUIT` from the real client, then terminate itself with exit code `0`.

Non-AUTH example

Assume we have a configuration file `conf.txt` and only one email transaction text file `email.txt` in `~/send`. The two files inherit the exemplars in the "File formations" section.

First, to set up the server, we can run in Bash

```
python3 server.py conf.txt
```

Wait a few second, we then set up the client in Bash.

```
python3 client.py conf.txt
```

Then the client will look into `~/send`, read and parse the email, the log the following

```
S: 220 Service ready
C: EHLO 127.0.0.1
S: 250 127.0.0.1
```

```
S: 250 AUTH CRAM-MD5
C: MAIL FROM:<bob@bob.org>
S: 250 Requested mail action okay completed
C: RCPT TO:<alice@example.com>
S: 250 Requested mail action okay completed
C: RCPT TO:<me@carol.me>
S: 250 Requested mail action okay completed
C: DATA
S: 354 Start mail input end <CRLF>.<CRLF>
C: Date: Mon, 14 Sep 1987 23:07:00 +1000
S: 354 Start mail input end <CRLF>.<CRLF>
C: Subject: Frist Electronic Mail from Bob to Alice and Carol
S: 354 Start mail input end <CRLF>.<CRLF>
C: Across the Exosphere we can reach every corner on the Moon.
S: 354 Start mail input end <CRLF>.<CRLF>
C: P.S. THIS IS NOT A SPAM
S: 354 Start mail input end <CRLF>.<CRLF>
C: .
S: 250 Requested mail action okay completed
C: QUIT
S: 221 Service closing transmission channel
```

On the other hand, the server should log exact same thing. In addition, an email transaction text `~/inbox/558623220.txt` should be saved before the server processes `QUIT` sent by the client. The content of the `~/inbox/558623220.txt` should be exactly the same as `~/send/email.txt` .

After sending the first and the only email in `~/send` , the client indeed should `QUIT` .

The server, however, should run indefinitely until receiving a `SIGINT` signal. If there are multiple children spawned by the server, all children need to be terminated gracefully as well.

AUTH example

With the same set up as the previous example, except the only email transaction text file is named `auth-email.txt` , the client will log the following

```
S: 220 Service ready
C: EHLO 127.0.0.1
S: 250 127.0.0.1
S: 250 AUTH CRAM-MD5
C: AUTH CRAM-MD5
S: 334 N2UyYjI1NTItMDI4Yy00ODE4LTkzNDctNDRmODRkOTRmNTE4
C: Ym9iIDQ3MTUwNGIxOTcwYzk0ZjJmYjQ4Y2E4ODkwY2Y1ZGRm
S: 235 Authentication successful
C: MAIL FROM:<bob@bob.org>
S: 250 Requested mail action okay completed
C: RCPT TO:<alice@example.com>
S: 250 Requested mail action okay completed
C: RCPT TO:<me@carol.me>
S: 250 Requested mail action okay completed
C: DATA
S: 354 Start mail input end <CRLF>.<CRLF>
C: Date: Mon, 14 Sep 1987 23:07:00 +1000
S: 354 Start mail input end <CRLF>.<CRLF>
C: Subject: Frist Electronic Mail from Bob to Alice and Carol
S: 354 Start mail input end <CRLF>.<CRLF>
C: Across the Exosphere we can reach every corner on the Moon.
S: 354 Start mail input end <CRLF>.<CRLF>
C: P.S. THIS IS NOT A SPAM
```

```
S: 354 Start mail input end <CRLF>.<CRLF>
C: .
S: 250 Requested mail action okay completed
C: QUIT
S: 221 Service closing transmission channel
```

On the other hand, the server should log the exact same thing. In addition, an email transaction text `~/inbox/auth.558623220.txt` should be saved before the server processes `QUIT` sent by the client. The content of the `~/inbox/auth.558623220.txt` should be exactly the same as `~/send/email.txt` .

After sending the first and the only email in `~/send` , the client indeed should `QUIT` . In the case there are more than one email text files, send them one at a time by dictionary order of the file names.

Eavesdropper example

The saving email behaviour eavesdropper should be same as a server. The logging behaviour is different from a server however stated clear before in the "Logging example". When testing the eavesdropper, make sure to launch the programs in the following order:

1. Real server
2. The eavesdropper
3. Real client

`SMTP-CRAM` Order of Events

Session Initiation

An SMTP session is initiated when a client opens a connection to a server and the server responds with an opening message. The server includes identification of the software and version information in the connection greeting reply after the `220` code. E.g.

```
S: 220 Service ready
C: EHLO 127.0.0.1
```

Client Initiation

Once the server has sent the welcoming message and the client has received it, the client normally sends the EHLO command to the server, indicating the client's identity.

In the EHLO command the host sending the command identifies itself; the command may be interpreted as saying "Hello, I am ".

Mail Transactions

There are three steps to SMTP mail transactions. The transaction starts with a MAIL command which gives the sender identification. In general, the MAIL command can be sent only when no mail transaction is in progress, otherwise an error will occur. A series of one or more RCPT commands follows giving the email recipient information. Then a DATA command initiates transfer of the mail data and is terminated by the "end of mail" data indicator, which also confirms the transaction.

Mail transaction commands MUST be used in the order discussed above.

Terminating Sessions and Connections

An SMTP connection is terminated when the client sends a QUIT command. The server responds with a positive reply code, after which it closes the connection.

An SMTP server MUST NOT intentionally close the connection except:

- After receiving a QUIT command and responding with a `221` reply.
- After detecting the need to shut down the SMTP service (E.g.,When the server receives a `SIGINT` signal) and returning a `421` response code. This response code can be issued after the server receives any command or, if

necessary, asynchronously from command receipt (on the assumption that the client will receive it after the next command is issued).

In particular, a server that closes connections in response to commands that are not understood is in violation of this specification. Servers are expected to be tolerant of unknown commands, issuing a `500` reply and awaiting further instructions from the client.

An SMTP server which is forcibly shut down via external means SHOULD attempt to send a line containing a `421` response code to all connected SMTP clients before exiting. The SMTP client will normally read the `421` response code after sending its next command.

SMTP clients that experience a connection close, reset, or other communications failure due to circumstances not under their control (in violation of the intent of this specification but sometimes unavoidable) SHOULD, to maintain the robustness of the mail system, treat the mail transaction as if a `451` response had been received and act accordingly.

Order restriction

There are restrictions on the order in which commands may be used.

A session that will contain mail transactions MUST first be initialized by the use of the `EHLO` command.

An `EHLO` command MAY be issued by a client later in the session. If it is issued after the session begins, the SMTP server MUST clear all buffers and reset the state exactly as if a `RSET` command had been issued. In other words, the sequence of `RSET` followed immediately by `EHLO` is redundant, but not harmful other than in the performance cost of executing unnecessary commands.

The `RSET` commands can be used at any time during a session, or without previously initializing a session. SMTP servers processes it normally (that is, not return a `503` code) even if no `EHLO` command has yet been received.

The `MAIL` command begins a mail transaction. Once started, a mail transaction consists of a transaction beginning command, one or more RCPT commands, and a `DATA` command, in that order. A mail transaction may be aborted by the `RSET` (or a new `EHLO`) command. There may be zero or more transactions in a session. `MAIL` command MUST NOT be sent if a mail transaction is already open, i.e., it should be sent only if no mail transaction had been started in the session, or it the previous one successfully concluded with a successful `DATA` command, or if the previous one was aborted with a `RSET` .

If the transaction beginning command argument is not acceptable, a `501` failure reply MUST be returned, and the SMTP server MUST stay in the same state. If the commands in a transaction are out of order to the degree that they cannot be processed by the server, a `503` failure reply MUST be returned, and the SMTP server MUST stay in the same state. E.g.:

- If a `RCPT` command appears without a previous MAIL command, the server MUST return a `503` “Bad sequence of commands” response.
- If there was no `MAIL` , or no `RCPT` command, or all such commands were rejected, the server returns a “command out of sequence” (`503`) reply in response to the `DATA` command.

The last command in a session MUST be the `QUIT` command. The `QUIT` command cannot be used at any other time in a session but be used by the client SMTP to request connection closure, even when no session opening command was sent and accepted.

`SMTP-CRAM` Request and Response

Application data is exchanged following a sequence of request-response messages.

Every client request must generate exactly one server response. Except the first client request, every client can only request again after receiving the server response regarding its previous request.

Request in general

A `SMTP-CRAM` request is consists of a command name and followed by parameters. The request command name is specified in the `<COMMAND>` . The parameters, if any, are separated by a single space after the command name. All fields are case-sensitive.

The maximum of a command line total length is 1024 bytes. This includes the command word, the parameter, and `<CRLF>` .

```
<COMMAND> [PARAMETER]...
```

In ABNF syntax:

```
REQUEST              =   COMMAND
                         *[ SP PARAMETER ]
                         CRLF
```

We add restrictions below:

```
COMMAND              =   "EHLO" /
                         ;; HELLO
                         "MAIL" /
                         ;; MAIL
                         "RCPT" /
                         ;; RECIPIENT
                         "RSET" /
                         ;; RESET
                         "NOOP" /
                         ;; NOOP
                         "QUIT" /
                         ;; QUIT
                         "AUTH" /
                         ;; AUTHENTICATE
PARAMETER            = *OCTET
```

Response in general

A `SMTP-CRAM` response is consists of a three-digit number (transmitted as three numeric characters) followed by some text unless specified otherwise in this document. The number can be used to determine what state the client is at; the text is for the human user. Exceptions are as noted elsewhere in this document. In the general case, the text is context dependent, so there are varying texts for each reply code. The response code is specified in the `<CODE>` . The parameters, if any, are separated by a single space after the command name. All fields are case-sensitive.

```
<CODE> [PARAMETER]...
```

In ABNF syntax:

```
RESPONSE             =   CODE
                         *[ SP PARAMETER *CRLF]
                         CRLF
```

We add restrictions below:

```
CODE                 =   "220" /
                         ;; Service ready
                         "221" /
                         ;; Service closing transmission channel
                         "235" /
                         ;; Authentication succeeded
                         "250" /
                         ;; Requested mail action okay, completed
                         "334" /
                         ;; Server BASE64-encoded challenge
                         "354" /
                         ;; Start mail input; end with <CRLF>.<CRLF>
                         "421" /
                         ;; Service not available, closing transmission channel
                         ;; (This may be a reply to any command if the service knows it must shut down)
                         "500" /
                         ;; Syntax error, command unrecognized
```

```
                        "501" /
                        ;; Syntax error in parameters or arguments
                        "503" /
                        ;; Bad sequence of commands
                        "504" /
                        ;; Command parameter not implemented
                        "535" /
                        ;; Authentication credentials invalid
PARAMETER           = *OCTET
```

Each command is listed with its usual possible replies. The prefixes used before the possible replies are "IM" for intermediate, "SC" for success, and "ER" for error. Note that error `421` can be responded to any commands if the server knows it is going to be terminated before the clients disconnect, therefore `421` is omitted.

```
CONNECTION ESTABLISHMENT
    SC: 220
EHLO
    SC: 250, 501
MAIL
    SC: 250
    ER: 501, 503
RCPT
    SC: 250
    ER: 501, 503
DATA
    IM: 354 -> data -> SC: 250
    ER: 501, 503
RSET
    SC: 250
    ER: 501
NOOP
    SC: 250
    ER: 501
AUTH
    IM: 334 -> data -> SC: 235
                       ER: 535
    ER: 501, 504
QUIT
    SC: 221
    ER: 501
```

Error code explanation

`500` : For the command name that cannot be recognized or the "request is too long" case. Note, if request is too long, it very likely that the longer-than-1024-bytes part cannot be recognized as a valid command.

```
500-response            = "500" SP "Syntax error, command unrecognized" CRLF
```

`501` : Syntax error in command or arguments. Commands that are specified in this document as not accepting arguments (`DATA` , `RSET` , `QUIT`) return a `501` message when arguments are supplied.

```
501-response            = "501" SP "Syntax error in parameters or arguments" CRLF
```

`503` : For a "Bad sequence of commands", previously discussed in "Order of commands" section.

```
503-response            = "503" SP "Bad sequence of commands" CRLF
```

`504` : This response to the `AUTH` command indicates that the authentication failed due to unrecognized authentication type.

```
504-response            = "504" SP "Unrecognized authentication type" CRLF
```

`535` : This response to the `AUTH` command indicates that the authentication failed due to invalid or insufficient authentication credentials.

```
535-response            = "535" SP "Authentication credentials invalid" CRLF
```

`EHLO`

`EHLO` example

```
S: 220 Service ready
C: EHLO 127.0.0.1
S: 250 127.0.0.1
S: 250 AUTH CRAM-MD5
```

`EHLO` request

The hello request.

After receiving `220` reply from the server, a client starts an SMTP session by issuing the `EHLO` command. The server will give either a successful response, a failure response, or an error response. In any event, a client MUST issue EHLO before starting a mail transaction. This command and a `250` reply to it, confirm that both the client and the server are in the initial state.

Log example:

```
C: EHLO 127.0.0.1
```

In ABNF syntax,

```
ehlo-request                = "EHLO" SP IPv4-address CRLF
```

`EHLO` response

The hello response.

The server will give either a successful response, or an error response. If `IPv4-address` is invalid, the server reports a `501-response` .

The successful response to `EHLO` request is a multiline reply. Each line starts with a successful code (`250`) followed by a space and other parameters separated by a space.

Log example:

```
S: 250 127.0.0.1
S: 250 AUTH CRAM-MD5
```

In ABNF syntax,

```
ehlo-ok-response            =   "250" SP IPv4-address
                                CRLF
                                "250" SP "AUTH" SP "CRAM-MD5"
                                CRLF
```

Where `IPv4-address` is the IPv4 address in the request, and not necessarily the address of the client.

Note, an `EHLO` command MAY be issued by a client later in the session. If it is issued after the session begins, the SMTP server MUST clear all buffers and reset the state exactly as if a `RSET` command had been issued. In other words, the sequence of `RSET` followed immediately by `EHLO` is redundant, but not harmful other than in the performance cost of executing unnecessary commands.

`MAIL`

`MAIL` example

```
C: MAIL FROM:<bob@example.org>
S: 250 Requested mail action okay completed
```

`MAIL` request

The mail request.

This command is used to initiate a mail transaction in which the mail data is delivered to an SMTP server. The argument field contains a source mailbox address (between "<" and ">" brackets).

This command clears the source buffer, the destination buffer, and the mail data buffer; and inserts the source information from this command into the source buffer.

Log example:

```
C: MAIL FROM:<bob@example.org>
```

In ABNF syntax,

```
mail-request                = "MAIL FROM:" "<" Source ">" CRLF
Source                      = Mailbox
Mailbox                     = Dot-string "@" Domain
Domain                      = (sub-domain 1*("." sub-domain)) / address-literal
sub-domain                  = Let-dig [Ldh-str]
address-literal             = "[" IPv4-address-literal "]"
Dot-string                  = Atom *("." Atom)
Atom                        = Let-dig *(Let-dig / "-")
```

`MAIL` response

The mail response.

The server will give either a successful response, or an error response. The server will send a `503-response` if the client is calling at a wrong state. See section " `SMTP-CRAM` state diagrams" for more information. If `Source` is invalid, the server reports a `501-response` .

If `Source` is valid, the server always returns a `250` reply.

Log example:

```
S: 250 Requested mail action okay completed
```

In ABNF syntax,

```
mail-ok-response            =   "250" SP "Requested mail action okay completed"
                                CRLF
```

`RCPT`

`RCPT` example

```
C: RCPT TO:<bob@example.org>
S: 250 Requested mail action okay completed
```

`RCPT` request

The recipient request.

This command is used to identify an individual recipient of the mail data; multiple recipients are specified by multiple uses of this command.

This command inserts the destination information from this command into the destination buffer.

Log example:

```
C: RCPT TO:<bob@example.org>
```

In ABNF syntax,

```
rcpt-request                = "RCPT TO:" "<" Destination ">" CRLF
Destination                 = Mailbox
Mailbox                     = Dot-string "@" Domain
Domain                      = (sub-domain 1*("." sub-domain)) / address-literal
sub-domain                  = Let-dig [Ldh-str]
address-literal             = "[" IPv4-address-literal "]"
Dot-string                  = Atom *("." Atom)
Atom                        = Let-dig *(Let-dig / "-")
```

`RCPT` response

The recipient response.

The server will give either a successful response, or an error response. The server will send a `503-response` if the client is calling at a wrong state. See section " `SMTP-CRAM` state diagrams" for more information. If `Destination` is invalid, the server reports a `501-response` .

If `Destination` is valid, the server always returns a `250` reply.

Log example:

```
S: 250 Requested mail action okay completed
```

In ABNF syntax,

```
rcpt-ok-response            =   "250" SP "Requested mail action okay completed"
                                CRLF
```

`DATA`

`DATA` example

```
C: DATA
S: 354 Start mail input end <CRLF>.<CRLF>
C: Never gonna
S: 354 Start mail input end <CRLF>.<CRLF>
C:  give you up
S: 354 Start mail input end <CRLF>.<CRLF>
C: .
S: 250 Requested mail action okay completed
```

`DATA` request

The data request.

This command is used to send mail data continuously until the end of mail data indication.

The server normally sends a `354` response to the client, and then the server treats the lines (strings ending in `<CRLF>` sequences) following the command as mail data from the client. The mail data is terminated by a line containing only a period, that is, the character sequence `<CRLF>.<CRLF>` . This is the end of mail data indication.

Log example:

```
C: DATA
C: Never gonna...
C: give you up
C: .
```

Note that the last client message should be exactly `<CRLF>.<CRLF>` .

In ABNF syntax,

```
data-request                = "DATA" CRLF
line-data                   = *OCTET CRLF
end-of-email                = CRLF "." CRLF
```

`DATA` response

The data response.

The server will give either a successful response, an intermediate response or an error response. The server will send a `503-response` if the client is calling at a wrong state. See section " `SMTP-CRAM` state diagrams" for more information. The server will send a `501-response` if one or more parameters are specified.

Otherwise, the server always continuously returns a `354` reply, until the end of email indicator is received, then finally returns a `250` reply.

Log example:

```
S: 354 Start mail input end <CRLF>.<CRLF>
S: 354 Start mail input end <CRLF>.<CRLF>
S: 354 Start mail input end <CRLF>.<CRLF>
S: 250 Requested mail action okay completed
```

In ABNF syntax,

```
data-int-response           =   "354" SP "Start mail input end <CRLF>.<CRLF>"
                                CRLF
data-ok-response            =   "250" SP "Requested mail action okay completed"
                                CRLF
```

`RSET`

`RSET` example

```
C: RSET
S: 250 Requested mail action okay completed
```

`RSET` request

The reset request.

This command specifies that the current mail transaction will be aborted. Any stored sender, recipients, and mail data MUST be discarded, and all buffers are cleared. A reset command may be issued by the client at any time.

Log example:

```
C: RSET
```

In ABNF syntax,

```
rset-request                = "RSET" CRLF
```

`RSET` response

The reset response.

The server will give either a successful response, or an error response. The server will send a `501-response` if one or more parameters are specified.

Otherwise, the server MUST send a `250` reply to a RSET command with no arguments.

Log example:

```
S: 250 Requested mail action okay completed
```

In ABNF syntax,

```
rset-ok-response            =   "250" SP "Requested mail action okay completed"
                                CRLF
```

`NOOP`

`NOOP` request

```
C: NOOP
S: 250 Requested mail action okay completed
```

`NOOP` request

The noop request.

This command does not affect any parameters, stored buffer or previously entered commands. It specifies no action other than that the receiver send an OK reply.

Log example:

```
C: NOOP
```

In ABNF syntax,

```
rset-request                = "NOOP" CRLF
```

`NOOP` response

The reset response.

The server will give either a successful response, or an error response. The server will send a `501-response` if one or more parameters are specified.

Otherwise, the server MUST send a `250` reply to a `NOOP` command with no arguments.

Log example:

```
S: 250 Requested mail action okay completed
```

In ABNF syntax,

```
noop-ok-response            =   "250" SP "Requested mail action okay completed"
                                CRLF
```

`AUTH`

The authentication protocol exchange consists of a series of server challenges and client answers that are specific to the authentication mechanism.

`CRAM-MD5` explanation

The server challenge is a presumptively arbitrary string.

The client answer is a string consisting of the user name, a space, and a digest. This is computed by applying the keyed MD5 algorithm from RFC 2104 HMAC: Keyed-Hashing for Message Authentication where the key is a shared secret. This shared secret is a string known only to the client and server. When the server receives this client response, it verifies the digest provided. If the digest is correct, the server should consider the client authenticated and respond appropriately.

`AUTH` example

Assume there is one valid user `bob` whose secret key is `postmanimpersonation`. Both `bob` and the server know this secret key. The server generates a random challenge `abfd32f6-a674-4589-b368-a1206d8be1f0`. The server callenge can then be BASE64 encoded to `N2UyYjI1NTItMDI4Yy00ODE4LTkzNDctNDRmODRkOTRmNTE4` and be sent to the client. The client BASE64 decodes the sent challenge and computes the answer, which contains a keyed MD5 digest of the challenge. The client produces a digest value (in hexadecimal) of the server challenge, which is `471504b1970c94f2fb48ca8890cf5ddf`. The user name `bob` is then prepended to it, forming `bob 471504b1970c94f2fb48ca8890cf5ddf`. The concatenation is then BASE64 encoded to form the client answer `Ym9iIDQ3MTUwNGIxOTcwYzk0ZjJmYjQ4Y2E4ODkwY2Y1ZGRm` and the client sends it back to the server. The server BASE64 decodes the answer, re-calculate the digiest because it knows the secret key of `bob`. Finally, it checks if the digest is correct.

You may find the Python standard library `hmac` being very helpful to compute the digest. Specifically, set `digestmod` to `md5`. To generate the one-time random server challenge (a.k.a. a nonce), a cryptographically secure pseudorandom number generator (CSPRNG) must be used. Please refer to the Python standard library `secrets` or the function `os.urandom()`.

Note, the below computation details are not the focus of the assignment. However, it presents to contribute the completeness of the specification.

The keyed MD5 digest is produced by calculating

```
MD5(
    (postmanimpersonation XOR opad),
    MD5(
        (postmanimpersonation XOR ipad),
        abfd32f6-a674-4589-b368-a1206d8be1f0
    )
)
```

where `ipad` and `opad` are as defined in the keyed-MD5 RFC 2104.

```
S: 220 Service ready
C: EHLO 127.0.0.1
S: 250 127.0.0.1
S: 250 AUTH CRAM-MD5
C: AUTH FOOBAR
S: 504 Unrecognized authentication type
C: AUTH CRAM-MD5
S: 334 N2UyYjI1NTItMDI4Yy00ODE4LTkzNDctNDRmODRkOTRmNTE4
C: Ym9iIDQ3MTUwNGIxOTcwYzk0ZjJmYjQ4Y2E4ODkwY2Y1ZGRm
S: 235 Authentication successful
```

```
S: 220 Service ready
C: EHLO 127.0.0.1
S: 250 127.0.0.1
S: 250 AUTH CRAM-MD5
C: AUTH CRAM-MD5
S: 334 N2UyYjI1NTItMDI4Yy00ODE4LTkzNDctNDRmODRkOTRmNTE4
C: Ym9iIGFiY2RlZmdoaWprbG1ub3BxcnN0dXZ3eHl6MTIzNDU2
S: 535 Authentication credentials invalid
```

Note, the second example fails the authentication because `Ym9iIGFiY2RlZmdoaWprbG1ub3BxcnN0dXZ3eHl6MTIzNDU2` can be BASE64 decoded to `bob abcdefghijklmnopqrstuvwxyz123456`. The digest `abcdefghijklmnopqrstuvwxyz123456` is clearly (ugly, hand-made and) incorrect.

`AUTH` request and response

The authenticate request and response.

The client will ask for a server challenge, then the server will give the challenge in an intermediate response. Then the client will send an answer to the challenge, and the server will give either a successful response or an error response.

The `AUTH` command indicates a SASL authentication mechanism to the server. In the scope of this assignment, only `CRAM-MD5` is implemented and can be accepted. Any other mechanism provided as an argument can be considered unsupported.

If the server supports the requested authentication mechanism (again, only `CRAM-MD5`), it performs an authentication protocol exchange to authenticate and identify the user. If the requested authentication mechanism is not supported, the server rejects the AUTH command with a `504` reply.

A server challenge is sent as a `334` reply with the first response parameter after code containing a BASE64 encoded string supplied by the SASL mechanism. The length of the server challenge needs to be greater than or equal to 16 bytes and less than or equal to 128 bytes. This challenge MUST NOT contain any data other than the BASE64 encoded challenge.

The client answer consists of a line containing a BASE64 encoded string. If the client wishes to cancel an authentication exchange, it issues a line with a single "*". If the server receives such an answer, it MUST reject the AUTH command by sending a `501` reply.

If the server cannot (BASE64) decode the argument, it rejects the AUTH command with a `501` reply. The server MUST reject invalid authentication answer data with a `535` reply. Should the client successfully complete the authentication exchange, the SMTP server issues a `235` reply.

The `AUTH` command is not permitted during a mail transaction. During a mail transaction, a server MUST reject any `AUTH` commands with a `503` reply. After an `AUTH` command has successfully completed, no more `AUTH` commands may be issued in the same session. After a successful `AUTH` command completes, a server MUST reject any further `AUTH` commands with a `503` reply.

In ABNF syntax,

```
auth-init-request           =   "AUTH" SP "CRAM-MD5" CRLF
auth-int-response           =   "334" SP auth-challenge
                                CRLF
auth-challenge              =   *OCTET
auth-client-answer          =   *OCTET CRLF
auth-ok-response            =   "235" SP "Authentication successful"
                                CRLF
```

Where `auth-challenge` is an abitrary long BASE64-encoded random string. It should not be static and not be shorter than 16 bytes.

In ABNF syntax, a successful authentication log example can be:

```
C: auth-init-request
S: auth-int-response
C: auth-challenge
S: auth-ok-response
```

In ABNF syntax, an unsuccessful authentication log example can be:

```
C: "AUTH" SP "CRAM-MD55555555" CRLF
S: 504-response
```

or

```
C: auth-init-request
S: auth-int-response
C: auth-challenge
S: 535-response
```

`QUIT`

`QUIT` example

```
C: QUIT
S: 221 Service closing transmission channel
```

Note that the client is expected to disconnect ONLY after receiving `221` .

`QUIT` request

The quit request.

This command specifies that the server MUST send an `OK` reply, and then close the transmission channel.

The client MUST NOT intentionally close the transmission channel until it sends a QUIT command and waits until it receives the `OK` reply, even if there was an error response to a previous command. The server MUST NOT intentionally close the transmission channel until it receives and replies to a valid `QUIT` command, even if there was an error caused by a previous command.

If the connection is closed prematurely due to violations of the above or system or network failure, the server MUST cancel any pending mail transaction, but not undo any previously completed transaction, and generally MUST act as if the command or transaction in progress had received a temporary error.

The `QUIT` command may be issued at any time.

Log example:

```
C: QUIT
```

In ABNF syntax,

```
rset-request                = "QUIT" CRLF
```

`QUIT` response

The quit response.

The server will give either a successful response, or an error response. The server will response a `501-response` if one or more parameters are specified.

Otherwise, the server MUST send a `221` reply to a QUIT command with no arguments.

Log example:

```
S: 221 Service closing transmission channel
```

In ABNF syntax,

```
rset-ok-response            =   "250" SP "Service closing transmission channel"
                                CRLF
```

`SMTP-AUTH` state diagrams

Minimal successful

The minimal state diagram for all successful operations is given below. We say it is minimal because this diagram currently ignores all the error handling plus some execute paths for the commands `NOOP` , `RSET` , `EHLO` and `QUIT` .

In this diagram, the even numbered states represent the situation when the server will send some response to the client, also when the client is about to receive some response. The odd numbered states represent the situation when the client has received a response and by which it has enough information to execute its next action and to send some request to the server. The text `C:...` and `S:...` on the arrow indicate what is sent by the client and server respectively. Note `S_3` repeats in the diagram, and the repeated state actually means the same state, however to make the demonstration clearer, the position is duplicated intentionally. (We want less cross-over for the arrows).

Most exceptional

The state diagram for most exceptional operations is given below. This diagram documents most error handling plus the ignored execute paths for the command `NOOP` , `RSET` , `EHLO` and `QUIT` in the previous section.

Note, `S:503` only applies to `MAIL` , `RCPT` and `DATA` for out-of-order requests.

`AUTH` exceptional

The state diagram for `AUTH` exceptional operations is given below. This diagram documents the error handling for `AUTH` that is missed in the previous section. E.g., `S:535` and `S:504` .

Self-testing

You are expected to write a number of test cases for server program ONLY. You are expected to test as many as possible execution paths of your code. Although you're asked to build the client already, implementing a server that can only handle your own client is not enough. The error conditions for the server are well documented in this specification, the server needs to robust and handle all the errors in the expected way.

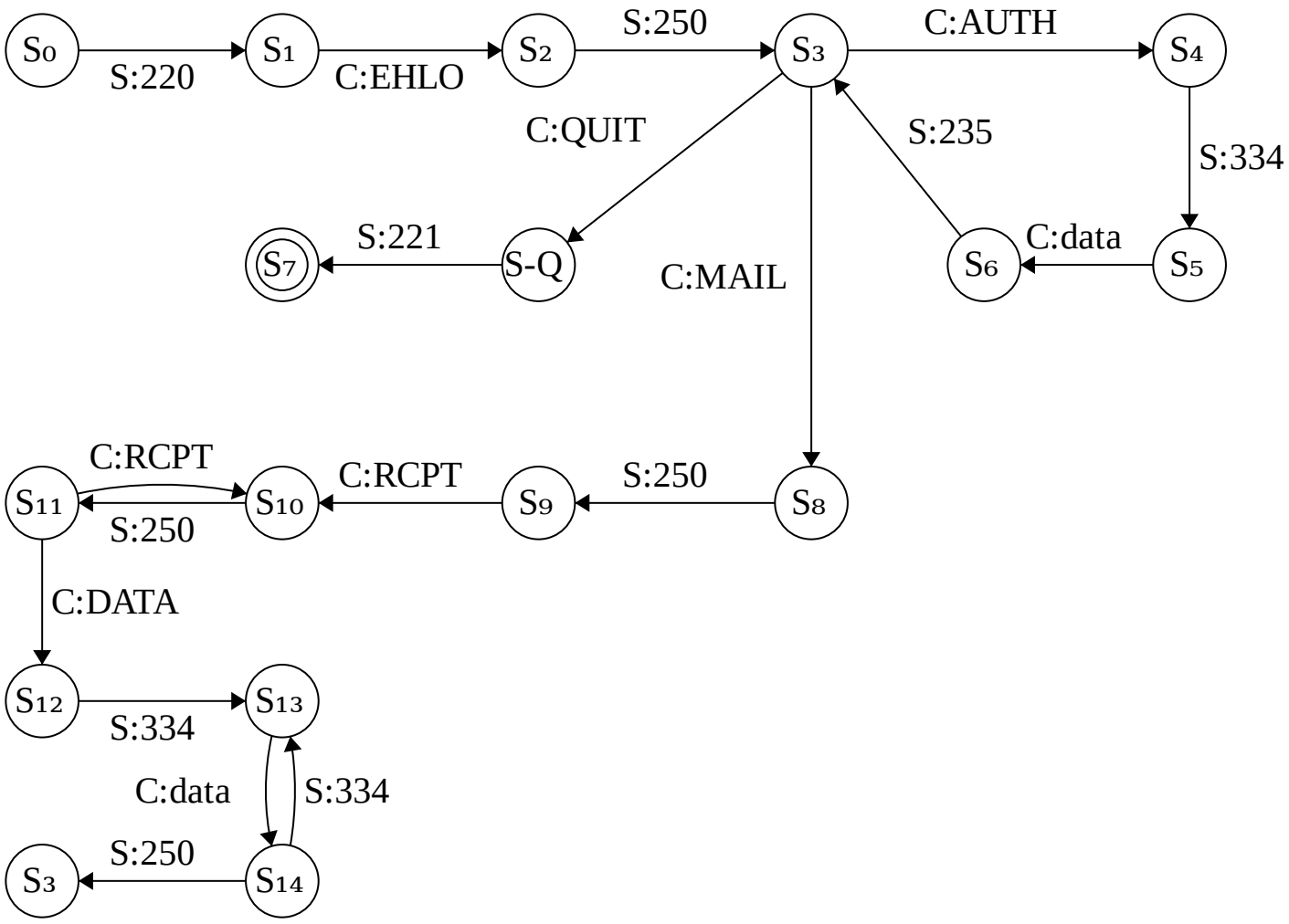

Figure 1: minimal

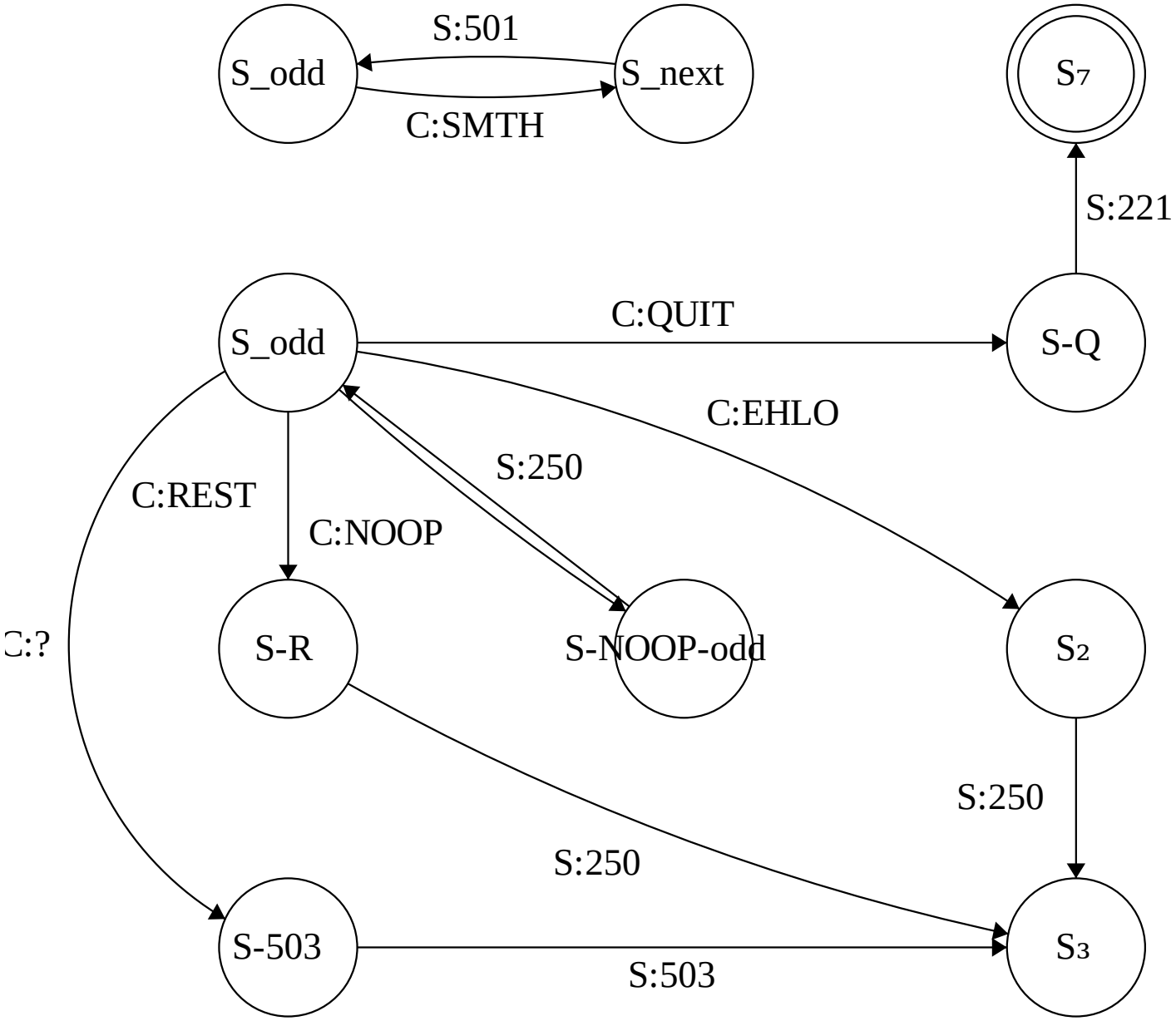

Figure 2: most-exceptional

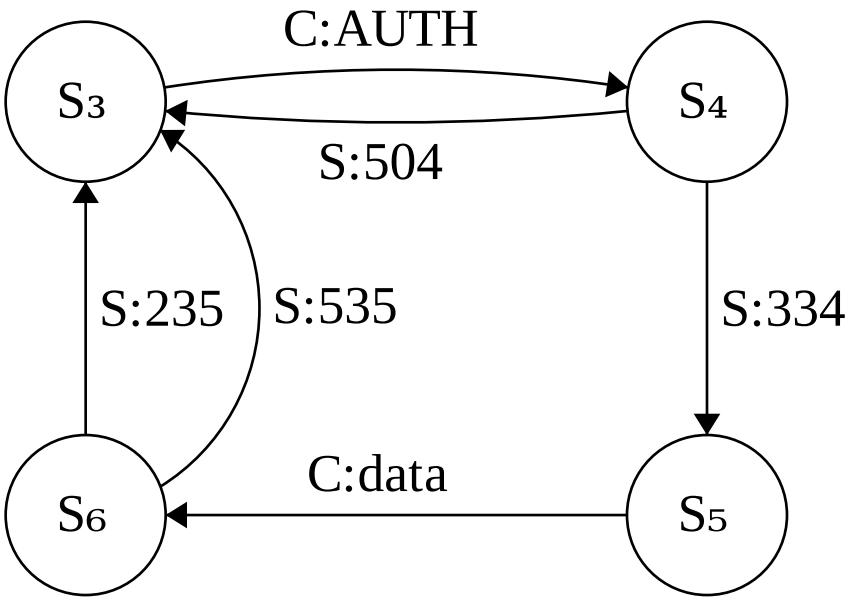

Figure 3: auth-exceptional

We will provide you with some test cases, but these do not test all the functionality described in the assignment. It is important that you thoroughly test your code by writing your own end-to-end (input/output) tests.

Unit tests are not required and not assessable.

For end-to-end testing, the input is the data (bytes encoded in ASCII, so plaintext) to be sent to the server, and the expected output is the log (`stdout`) of the server.

You must place all your end-to-end test cases in the `e2e_tests/` directory. Ensure that each pair of test cases must have an `<name>.in` input file and an `<name>.out` output file. Your own tests will not be marked if this naming convention is not followed. We recommend that the names of your test cases are descriptive so that the reader can easily know what each case is testing. E.g., `EHLO_01.in` , `AUTH_02.in` .

`Coverage.py` will be used to generate the coverage of your own tests and the coverage rate will determine some of the testing marks. Unix tools like `netcat` and `telnet` can help test your program.

Implementation

The assignment is to be implemented in Python. A set of scaffold files will be provided. You are expected to write legible code with good style, E.g.,PEP 8.

Here is a list of initially approved modules:

```
typing,
os,sys,pathlib,
regex,
date,time,datetime,
socket,
hmac,hashlib,secrets
```

Here is a list of blacklisted modules that must not be allowed even upon request.

```
configparser,
email,smtplib,smtpd,aiosmtpd,
threading,multiprocessing,subprocess
```

You are **NOT** allowed to import any Python modules that are not in the list of approved modules. You are free to use all built-in functions of Python. You are **NOT** allowed to include third-party Unix program directly (E.g., `os.system('netcat')` or equivalence) in your program.

If you want to use an additional module which will not trivialize the assignment, please ask on Ed. The approved library list may be extended. Related announcement should be posted accordingly.

May you have doubts on whether certain actions conducted in your solution are allowed or not, please ask on Ed.

You should **NOT** preserve any data in-progress temporarily on disk, but in the memory (this does not include the email files servers which will be saved on disk).

Please be mindful: breaching the above restrictions will result in harsh deduction for the entire assignment.

Submitting your code

An Ed Lesson workspace will be available for you to test and submit your code. Public test cases will be released up to **Week 12, Thursday, October 26th**. Additionally, there could be a set of unreleased test cases which will be run against your code after the due date.

You will need to use `git` to submit your code. To learn what `git` is and how to setup `git` (and SSH key) properly, please refer to Lecture 4.

Where to start

Note: the following numbering does not necessarily suggest the ordering that you should consider them.

1. Review the lab content for Week 8 and 10, and Homework 5 Task "Python Echo Server". Do research on `socket` :
   - socket documentation.
   - Socket Programming HOWTO.

   Find good TCP `socket` programs on the Internet (not only `StackOverflow`) and learn from the exemplars. Think about what makes `socket` program harder to debug, and first build your own debugging pipeline. E.g., Develop a pair of minimal viable client and server that can log all transaction chronologically. Eventually you will need it for your assignment.
2. Write a parser to verify the email transaction text format.
3. First the client, then the server, finally the eavesdropper. The client is easier to implement than the server. The behaviour of the client is less complex, whereas the server needs to be robust and handle any client - not limited to the one you develop. It would be almost impossible to have a working eavesdropper if the client and the server are not finished.
4. Consider what states and buffers mean on the server side. Why they are important? How can client behaviours affect the data structure that the server holds? How can you code this stateful server? E.g., define your own states and manage them.
5. The forbidden libraries involve existing `SMTP` implementations available in Python. You cannot use them or copy their source code directly, but for certain design, you may refer to them. E.g., state management.

Marking Criteria

The assignment will be marked with an automatic testing system on Ed. A mark will be given based on

- The tests passed for Task 1, 2 and 3 (16%). However, manual deduction may be applied if specific implementation requirements are not followed.
  - Task 1 (6%).
  - Task 2 (6%).
    - Use `CSPRNG` to generate challenge.
  - Task 3 (4%).
    - `fork` implementation.
- Self-testing (2%).
  - You are expected to write your own tests and submit them with your code, and a mark will be given based on coverage and manual inspection of your tests.
  - Your test cases must cover as many as possible execution paths and designed to test specific features of the server.
- A manual mark will be given to your report in Task 4 (2%).

A bonus mark (2%) may be given after completion of the first three tasks. As a condition, you will NOT be able to receive bonus marks unless all public test cases in Task 1, 2 and 3 have been completed. For requirements, please refer to the "Multi-process server" section. The mark will be awarded based on both passed automatic tests and a manual inspection.

There will be public test cases made available for you to test against, but there will also be extra non-public tests used for marking. Success with the public tests doesn't guarantee your program will pass the private tests.

Task 4

Write a report **IN YOUR OWN WORDS**:

- Define authentication, authorization and encryption using real world examples. In less than 150 words.
- Discuss what the `SMTP-CRAM` protocol can and cannot offers in terms of the above three concepts. If it can support one aspect, briefly describe the mechanism; if it fails to support one aspect, give a brief counterexample to support. In less than 150 words.
- Cite all references clearly. No citation format requirement.

Friendly note and important dates

Sometimes we find typos or other errors in specifications. Sometimes the specification could be clearer. Students and tutors often make great suggestions for improving the specification. Therefore, this assignment specification may be clarified up to **Week 12, Thursday, October 26th**. No major changes will be made. Revised versions will be clearly marked, and the most recent version announced to the class via Ed Discussions.

Before making a post on Ed, please search the keywords to find if there is any identical or related post. Duplication of identical post is not encouraged.

When making a post, we encourage you to make the post "public" if it is not personal or confidential. Everyone can benefit from a "public" post however not a "private" one. You can make an "anonymous public" post if you don't want to reveal your identity.

Warning

Any attempts to share your confidential personal credentials will be investigated. If the indentional act of exchanging credentials is confimred, it will result in an immediate zero for the entire assignment. This applies to any parties that participate in the exchange intentionally.

Any attempts to deceive the automatic marking system will result in an immediate zero for the entire assignment.

Negative marks can be assigned if you do not properly follow the assignment description, or your code is unnecessarily or deliberately obfuscated.

Academic Declaration

By submitting this assignment, you declare the following:

I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

Changes

Major changes will be announced at here.