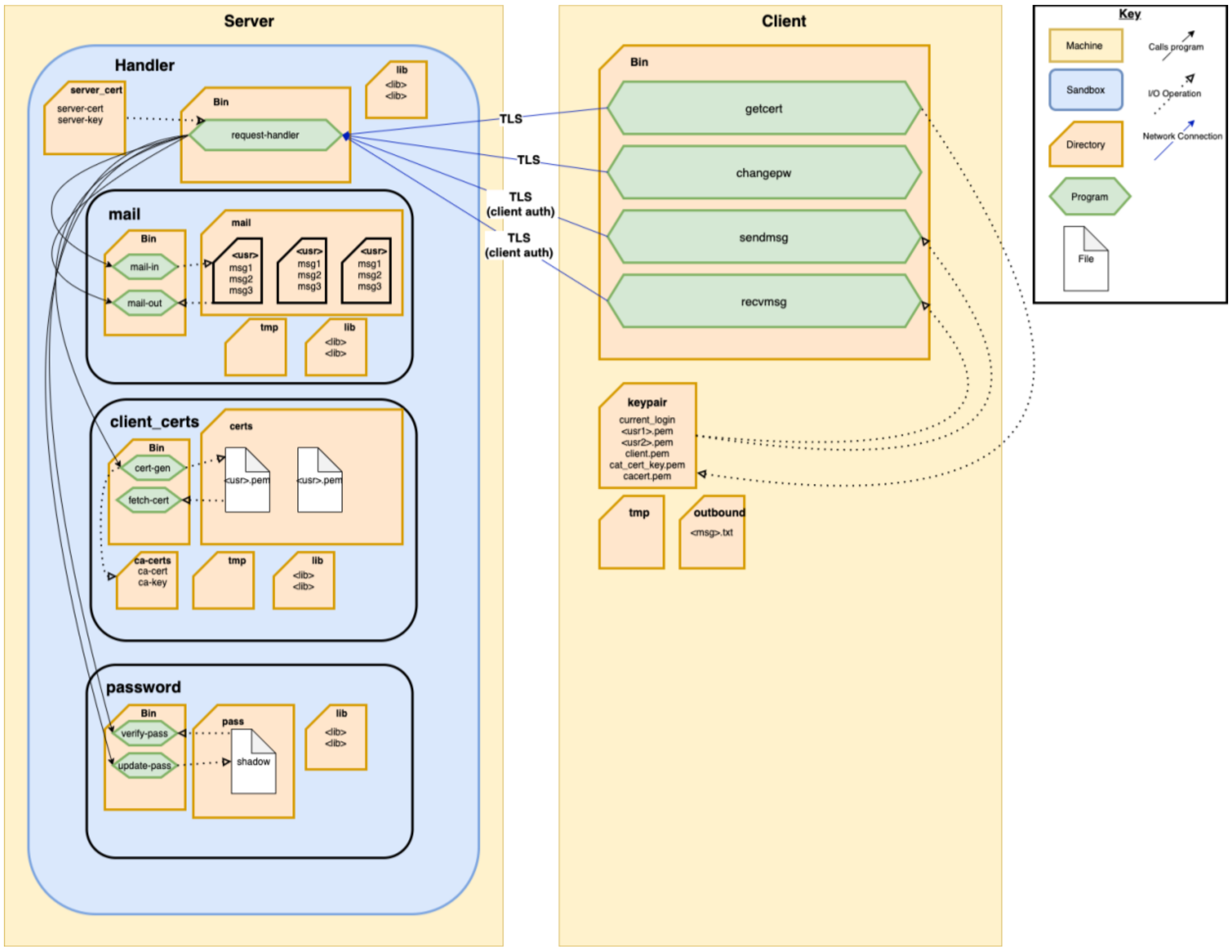
COMS4181 Final Project Architecture

Group: Samuel Deng (sd3013) and Francis Hogan (fwh2110)

A. Overall Architecture

In this section, we detail the components of our architecture and how they interact in a component diagram. For a larger picture, please see the end of this document.



In the above diagram, the **Server** box represents the necessary filesystem on the server's side, and the **Client** box represents the necessary filesystem on the client's side. We note that the anticipated testing strategy is roughly as follows:

1. The tester sets up the **system**. This installs the **Server** filesystem on a virtual machine containing the main server program **request-handler.** This also installs the **Client** filesystem and its four commands: **getCert, changePW, sendMsg,** and **rcvMsg** (see **C. Client Programs**).
2. The tester starts the server by running the **request-handler** program with root privileges.

3. The tester opens a new ssh connection on the same virtual machine to run the client apps to communicate with the running server (see **C. Client Programs**).

For how the sandboxes are implemented, see **D. Security - Sandboxing.** For how the privileges and permissions on the system are doled out, see **E. Security - File Permissions.**

B. Server Programs

In this section, we detail the programs required by our architecture on the server's side. The basic structure is that the server has one overall handler program **request-handler** which listens for client requests and invokes the corresponding internal server program: **mail-in, mail-out, cert-gen, verify-pass, update-pass.** These internal server programs all operate in their own three secure sandboxes: mail, passwords, and certs (more on this in **D. Security - Sandboxing**).

request-handler

Input: None (user is asked for password for the server private key).
Output: None (listens until Ctrl+C).
Responds To:
- **getcert**
- **changepw**
- **sendmsg**
- **recvmsg**

Invokes:
- **mail-in**
- **mail-out**
- **cert-gen**
- **fetch-cert**
- **verify-pass**
- **update-pass**

Resources:
- *server_certs/* (directory)

Privileges: Server (like the Apache webserver example) starts as root and sheds privileges. It needs root in order to invoke chroot and sandbox each individual server program. Permissions on all helper programs: **mail-in, mail-out, cert-gen, fetch-cert, verify-pass, update-pass**. **request-handler** does directly access the *mail/, client_certs/*, or *passwords/* directories directly. Because of this, it *must* call the helper programs (**verify-pass, update-pass, mail-in, mail-out, cert-gen**) in order to accomplish its desired tasks.

Program Overview: The workflow of the **request-handler** program is as follows:
1. **request-handler** is fired up and starts listening on the port specified by its inputs.

2. Unless the process is killed, **request-handler** waits for the following client requests:
    a. **getcert:** Client sends over *CSR*, *username*, and *password*. A standard TLS connection (not client-authenticated) is established. Then, the handler process makes invokes two programs, in this order:
        i. **verify-pass.** Handler passes client's *username* and *password* to **verify-pass.** If this fails, do not go on to step (ii).
        ii. **cert-gen**. Handler passes client's *username* and *CSR* to **cert-gen** and **cert-gen**, if successful, stores the cert and returns it back to **request-handler,** which sends it back to the client.
    b. **changepw:** Client sends over *CSR, username*, *oldPassword*, and *newPassword*. A standard TLS connection (not client-authenticated) is established. The handler invokes four programs, in this order:
        i. **verify-pass.** Handler passes client's *username* and *oldPassword* to **verify-pass.** If this fails, do not go on to step (ii).
        ii. **mail-out.** Handler passes client's *username* to **mail-out** which checks if there are any pending messages still. **mail-out** exits with a *msgFound* flag. If *msgFound* is false, fail and do not continue to step (iii).
        iii. **update-pass.** Handler passes client's *username,* and *newPassword* to **update-pass,** which updates the password file.
        iv. **cert-gen.** Handler passes client's *username* and *CSR* to **cert-gen,** which stores the cert and returns it back to **request-handler**, which sends it back to the client.
    c. **sendmsg:** Client authenticates via TLS with *clientCertificate* (this is a client-auth connection), and a list *recipientNames.* Then, the handler:
        i. Establishes a TLS connection with the *clientCertificate* and its own *serverCertificate* in the *server_certs/* directory. This is a Client-Authenticated TLS handshake (as opposed to the standard TLS handshake where only a server certificate is used).
        ii. The client's *username* is extracted from *serverCertificate* (it needs to be in the client cert's *commonName* field).
        iii. **fetch-cert.** Handler passes the *recipientNames* to **fetch-cert** which sends to the client the encryption certificates for each name in *recipientNames.* If any of the recipients are invalid, the client is notified which are invalid and needs to retry the request with a fully valid recipient list.
        iv. After the client sends back the encrypted, signed *message* the handler calls **mail-in**, once each for each *message* in *messages* (encrypted by recipient, signed by client) and *recipient* in *recipientNames*, so **mail-in** writes the message in the inboxes for each of the designated recipients in *recipientNames.*
    d. **recvmsg:** Client authenticates via TLS with *clientCertificate* (this is a client-auth connection). Then, the handler:
        i. Establishes a TLS connection with *clientCertificate* and its own *serverCertificate* in the *server_certs/* directory. This is a Client-

Authenticated TLS handshake (as opposed to the standard TLS handshake where only a server certificate is used).

   ii. **mail-out.** Handler calls **mail-out** which exits with a *msgFound* flag. If *msgFound* is false, then return a predetermined message indicating no messages found for the client. Else, if *msgFound* is true, we continue (NOTE: signature is from the original sender, the encryption is from the current recipient's encryption certificate).
   iii. **fetch-cert.** Handler calls **fetch-cert** with the single username passed back from mail-out as a *sender* field. The handler receives a single cert from **fetch-cert.**
   iv. Finally, **request-handler** sends back the *msg* and the *cert* from **fetch-cert** back to the client.

mail-in

Input: *message* (encrypted by recipient, signed by client), *recipient.*
Output: None (but writes the messages in inboxes for each of designated *recipientNames*). Return 0 for success, 1 for failure.
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *mail/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read, write and execute on each *user* directory in the *mail/* directory. This is so **mail-in** can write mail messages to the directory. However, it has no privileges on the messages after they are written (not even read directories), because there is no need for message modification.

Program Overview: The workflow of the **mail-in** program is as follows:

1. For the given *recipient*:
   a. Write the encrypted, signed *message* in the corresponding index to the appropriate mailbox.
      i. Write a message with two parts (newline delimited): (1) sender (2) signed, encrypted message.
      ii. The filename is a simple numerical index indicating when the message was delivered. This will always be iterated in ascending order.

mail-out

Input: *username* (the specific user whose inbox we would like to check for mail). Takes in a message (signed, encrypted) as stdin.
Output: Exits with one of three flags: *EMPTY, ERROR,* or *MSG_FOUND.*
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *mail/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read, write and execute on each *user* directory in the *mail/* directory. This is so **mail-out** can access mail messages to the directory. For the individual message files, **mail-out** has read permission, so it can eventually output the encrypted messages to the client if needed.

Program Overview: The workflow of the **mail-out** program is as follows:

1. For the specified *username*, check if there are any pending messages in their directory.
2. If there is a pending message:
    a. Read the EARLIEST message (each message's name is indexed iteratively) and set to the message file the signed, encrypted message. The first line is the sender of the message, the second line is the encrypted message.
    b. Return the *message* which is formatted with (1) a sender (2) a message (back to **request-handler**).
    c. Delete this message from the directory.
    d. Exit with the *MSG_FOUND.*
3. Else (no pending messages):
    a. Exit with the *EMPTY* flag to the **request-handler.**

cert-gen

Input: *username, CSR* (both for the specific user the client logs in as).
Output: Return 0 for success, 1 for error. The *cert* is written as a temporary file.
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *client_certs/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read, write and execute on each *user* directory in the *client_certs/* directory. This is so **cert-gen** can access the actual certificates in the directory. For the individual certificate files, **cert-gen** has write permission, so it can update the client certificates as needed.

Program Overview: The workflow of the **cert-gen** program is as follows:

1. Using the provided *CSR*, use the certificate API to generate a new certificate for the user.
2. Write these new certificates to the directory of *username*.
3. Return back to **request-handler** the *cert* for the client.

fetch-cert

Input: *recipient*.
Output: Return 0 for success, 1 for failure. *Cert* is written to a temporary file for **request-handler** to read.
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *client_certs/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read, write and execute on each *user* directory in the *client_certs/* directory. This is so **fetch-cert** can access the actual certificates in the directory. For the individual certificate files, **fetch-cert** has read permission, so it can get each appropriate encryption cert.

Program Overview: The workflow of the **fetch-cert** program is as follows:

1. Check if *recipient* exists.
    a. If *recipient* does exist, then read its cert and write to a temporary file.
2. Signal to **request-handler** to read this temporary *cert* file to get the cert.

verify-pass

Input: *username* (user we need to verify) and *password.*
Output: Return 0 for success (verified), 1 for error (cannot verify).
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *password/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read the *shadow* file in the *password/* directory. This is so **verify-pass** can read the hashed passwords in the directory. For the individual password file, **fetch-cert** has read permission, so it can read and compare.

NOTE: In this design, we assume that no new users can be added to the system, and our setup scripts will install the file system such that the 35 users on the system (the ones that were on HW 3). Their passwords are given in *shadow.*

Program Overview: The workflow of the **verify-pass** program is as follows:

1. Read the *password* in the *username*'s password directory.
2. Hash *password* with SHA512 and check against *shadow* which contains the salted, hashed passwords.

3. Compare to *password* and return corresponding boolean.

update-pass

Input: *username* and *newPassword.*
Output: Return for 0 success and 1 for failure.
Responds To:

- **request-handler**

Invokes:

- **N/A**

Resources:

- *password/* (directory)

Privileges: ONLY executable by **request-handler** (no other users can execute this program). Able to read the *shadow* file in the *password/* directory. This is so **verify-pass** can access the hashed passwords in the directory. For the individual password files, **fetch-cert** has write permission, so it can overwrite the password file.

Program Overview: The workflow of the **update-pass** program is as follows:

1. Open and update the existing password file for the user *username* using the *newPassword* input (which is already hashed from the client's program).
    a. This is done by salting and doing a new SHA512 hash.

C. Client Programs

In this section, we detail the programs required by our architecture on the client's side. As detailed in the spec, the client has access to four programs: **getcert, changepw, sendmsg,** and **recvmsg.** Each of these programs communicate with the server via TLS, where **getcert** and **changepw** have a standard TLS connection and **sendmsg** and **recvmsg** have a client-verified TLS connection. One thing to note for each of these programs is that the *keypair/* folder on the client's side is protected via permissions from users that are not these four programs, and the specific permissions are detailed below.

getcert

Input: *username* (*password* is asked for via system prompt).
Output: Program just writes to the *keypair/* directory on the client side).
Privileges: Able to read, write, and execute the client's *keypair/* directory. For the individual files in *keypair/*, **getcert** has write access.

Program Overview: The workflow of the **getcert** program is as follows:

1. Prompt user to enter a *password* via getpass().
2. Establish a TLS connection (not client-authenticated) with the server.

3. Send over the *username* and *password*, along with *CSR* generated by this **getcert** program.
4. Receive a certificate, *cert*, from the server, with indication of whether it is new or the existing certificate.
5. Write the certificate to the *certs/* directory on the client's side, either keeping the current cert or making a new one (if it doesn't already exist).
6. Write to the *current_login* file the current *username.* The system will only work for this user until a new **getcert** or **changepw** command is called.

HTTPS Command:
<verb>: POST
<url>: https://localhost/getcert
<version>: HTTP/1.0
<option-lines>:

- Content-Length: <bytes>

body:
username
password
CSR (certificate signing request)

changepw

Input: *username* (passwords are asked for at system prompt).
Output: Program just writes to *keypair/* on the client side.
Privileges: Able to read, write, and execute the client's *keypair/* directory. For the individual files in *keypair/*, **getcert** has write access.

Program Overview: The workflow of the **changepw** program is as follows:

1. Prompt user to enter its CURRENT password, call it *oldPassword.*
2. Prompt the user to enter its NEW password, call it *newPassword.*
3. Establish a TLS connection (not client-authenticated) with the server.
4. Send over the *username, oldPassword*, and *newPassword* along with *CSR* newly generated.
5. Receive a certificate, *cert*, from the server.
    a. This might not occur if it happens that the client still has pending messages. In this case, an appropriate message will be shown to the client and an error response is sent by the server.
6. Write the certificate to the *keypair/* directory on the client's side, updating the existing cert.

HTTPS Command:
<verb>: POST
<url>: https://localhost/changepw
<version>: HTTP/1.0

<option-lines>:

- Content-Length: <bytes>

body:
username
old password
new password
CSR (certificate signing request)

sendmsg

Input: *recipientNames* (passed as arguments), *message* (passed as a path in the arguments). The *message* must belong to the *outbound/* folder (the only folder the client has write access in on the client's side).
Output: None (program just eventually sends encrypted messages back to the server).
Privileges: Able to read and execute the client's *keypair/* directory. Able to read/write to *tmp/.* For the individual files in *keypair/*, **getcert** has read access.

Program Overview: The workflow of the **sendmsg** program is as follows:

1. Read the client certificate in the *keypair/* directory.
2. Establish a TLS connection (client-authenticated) with the server.
    a. This requires client to send over its correct client certificate, which may only occur when the user is logged in (this occurs in **getcert** or **changepw**).
3. Send over the *recipientNames.*
4. Receive from the server a list of encrypted certificates, call this *encryptCerts.*
5. Start a list called *sendRecipients* and a list called *encryptedMessages.* For each *cert* in *encryptCerts*:
    a. Sign the message with the current logged in user's certificate in *keypair/* directory.
    b. Encrypt the message with the corresponding *cert* (for each corresponding recipient) and append the encrypted message to *encryptedMessages*.
    c. Append the *recipient* to *sendRecipients.*
6. Send back to the server *sendRecipients* and *encryptedMessages*.

NOTE: By design, the **sendmsg** command will only send if ALL the recipients are valid on the mail system. If not, none of the messages will be sent but the client will be notified exactly which recipients are invalid so they can fix their request and retry. All this information is provided by the server for availability purposes.

HTTPS Command #1 (Get Encryption Certs):
<verb>: POST
<url>: https://localhost/sendmsg-encrypt
<version>: HTTP/1.0
<option-lines>:

- Content-Length: <bytes>

body:
recipient name 1
recipient name 2
.
.
.
recipient name n

HTTPS Command #2 (Send the message):
<verb>: POST
<url>: https://localhost/sendmsg-message
<version>: HTTP/1.0
<option-lines>:

- Content-Length: <bytes>

body:
recipient name 1:recipient name 2:...:recipient name n (colon delimited)
encrypted message 1
encrypted message 2
.
.
.
encrypted message n

recvmsg

Input: None (client certificate is read during program execution).
Output: Message is displayed in stdout.
Privileges: Able to read and execute the client's *keypair/* directory. Able to read/write to *tmp/.* For the individual files in *keypair/*, **getcert** has read access.

Program Overview: The workflow of the **recvmsg** program is as follows:

1. Read the client certificate in the *keypair/* directory.
    a. The username from the client sending this request is read from the CommonName field in the certificate. Therefore, it must pass client authentication to even get to receiving a message (it must have a certificate from the server in the first place).
2. Establish a TLS connection (client-authenticated) with the server.
3. Receive from the server a single encrypted message, *message*, and the *cert* from the original sender of the message (needed for verification).
4. Decrypt the message using the client's secret key.
5. Verify the signature on the message matches (via sender's own *cert* in the *keypair/* directory on the client side, which only can occur if the client is logged in).
6. Print the message to stdout.

HTTPS Command:
<verb>: GET
<url>: https://localhost/recvmsg
<version>: HTTP/1.0
body (empty):

D. Security - Sandboxing

- Sandboxes
    - **Handler:** The server consists of a single top-level sandbox that contains the main server executable (**request-handler**), the *server_cert/* directory, and three other sandboxes:
        - **Mail:** Within the top-level sandbox is a sandbox that is responsible for everything related to mail on the server. It contains the **mail-in** and **mail-out** executables as well as a mailbox for each user containing that users unreceived (unread) messages.
        - **Certs:** Within the top-level sandbox is a sandbox that is responsible for everything related to certificates on the server. It contains the **cert-gen** and **fetch-cert** executables as well as a folder for each user containing that user's tls certs and a folder containing the root CA and intermediate CA certificates.
        - **Pass:** Within the top-level sandbox is a sandbox that is responsible for everything related to passwords on the server. It contains the **verify-pass** and **update-pass** executables as well as a folder for each user containing the user's hashed passwords.
- Reasoning
    - Everything is contained in a top-level sandbox to isolate the mail system as a whole from anything else that may also run on the same server machine.
    - Within the top-level sandbox the mail system is divided into three sandboxes. The three types of content the server maintains are client certificates, client passwords (hashed), and client mailboxes. We isolated these three types of content along with the executables that interact with that content into separate sandboxes. This will help to reduce the "reach" that the various server executables have on the server by only giving them access to the content they need and dividing responsibilities of the server across different sandboxes.
- Implementation
    - To isolate programs in the system to run in separate sandboxes we utilize the chroot functionality provided by UNIX.
    - The install script sets up each sandbox environment with the appropriate file structure, executables, and libraries used by those executables.
        - Running the ldd command on each executable provides us with the libraries needed for that executable to run. We copy those libraries in from the main lib folders of the machine into the lib folders of the sandbox

    - The code utilizes the chroot and the execl apis to run the different programs in the created sandboxed environments.

E. Security - Users, Owner & Permissions

Sever

- Users
    - mail-in-usr
    - mail-out-usr
    - cert-gen-usr
    - fetch-cert-usr
    - verify-pass-usr
    - update-pass-usr
- Owners and permissions
    - *bin* folder in top-level sandbox
        - Owner: root
        - User:: r - x
        - Group:: - - -
        - Other:: r - x
    - **request-handler**
        - Owner: root
        - User:: - - x
        - Group:: - - -
        - Other:: - - -
    - *server_cert* (dir)
        - Owner: root
        - User:: r - x
        - Group:: - - -
        - Other:: - - -
    - server-cert.pem
        - Owner: root
        - User:: r - -
        - Group:: - - -
        - Other:: - - -
    - server-key.pem
        - Owner: root
        - User:: r - -
        - Group:: - - -
        - Other:: - - -
    - *lib* folders in top-level sandbox

    - Owner: root
    - Group: root
    - User:: r w x
    - Group:: r - x
    - Other:: r - x
- *bin* folder in mail sandbox
    - Owner: root
    - User:: r - x
    - Group:: - - -
    - Other:: - - -
- mail-in
    - Owner: mail-in-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- mail-out
    - Owner: mail-out-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- *mail* folder in mail sandbox
    - Owner: mail-in-usr
    - User:: r w x
    - User:mail-out-usr: r w x
    - Group:: - - -
    - Other:: - - -
- *user* folders in mail folder (user mailboxes)
    - Owner: mail-in-usr
    - User:: r w x
    - User:mail-out-usr: r w x
    - Group:: - - -
    - Other:: - - -
- *lib* folders in mail sandbox
    - Owner: root
    - Group: root
    - User:: r w x
    - Group:: r - x
    - Other:: r - x
- *tmp* folder in mail sandbox
    - Owner: mail-in-usr

    - User:: r w x
    - User:mail-our-usr: r w x
    - User:root: r - x
    - Group:: - - -
    - Other:: - - -
- *bin* folder in client_certs sandbox
    - Owner: root
    - User:: r - x
    - Group:: - - -
    - Other:: - - -
- **cert-gen**
    - Owner: cert-gen-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- **fetch-cert**
    - Owner: fetch-cert-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- *certs* folders in client_certs sandbox
    - Owner: cert-gen-usr
    - User:: r w x
    - User:fetch-cert-usr: r - x
    - Group:: - - -
    - Other:: - - -
- *ca-certs* folder in client_certs sandbox
    - Owner: cert-gen-usr
    - User:: r - x
    - Group:: - - -
    - Other:: - - -
- cacert.pem
    - Owner: cert-gen-usr
    - User:: r - -
    - Group:: - - -
    - Other:: - - -
- ca.key.pem
    - Owner: cert-gen-usr
    - User:: r - -
    - Group:: - - -

    - Other:: - - -
- *lib* folders in client_certs sandbox
    - Owner: root
    - Group: root
    - User:: r w x
    - Group:: r - x
    - Other:: r - x
- *tmp* folder in client_certs sandbox
    - Owner: cert-gen-usr
    - User:: r w x
    - User:fetch-cert-usr: r w x
    - User:root: r - x
    - Group:: - - -
    - Other:: - - -
- *bin* folder in password sandbox
    - Owner: root
    - User:: r - x
    - Group:: - - -
    - Other:: - - -
- **verify-pass**
    - Owner: verify-pass-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- **update-pass**
    - Owner: update-pass-usr
    - User:: - - -
    - User:root: - - x
    - Group:: - - -
    - Other:: - - -
    - setuid bit set
- *pass* folders in password sandbox
    - Owner: update-pass-usr
    - User:: r - x
    - User:verify-pass-usr: r - x
    - Group:: - - -
    - Other:: - - -
- shadow file in user folders
    - Owner: update-pass-usr
    - User:: r w -
    - User:verify-pass-usr: r - -
    - Group:: - - -

        - Other:: - - -
    - *lib* folders in password sandbox
        - Owner: root
        - Group: root
        - User:: r w x
        - Group:: r - x
        - Other:: r - x

Client

- Users
    - get-cert-usr
    - auth-cert-usr
- Owners and permissions
    - *bin* folder
        - User:: - - -
        - Group:: - - -
        - Other:: r - x
    - **getcert**
        - Owner: get-cert-usr
        - User:: - - x
        - Group:: - - -
        - Other:: - - x
        - setuid bit set
    - **changepw**
        - Owner: get-cert-usr
        - User:: - - x
        - Group:: - - -
        - Other:: - - x
        - setuid bit set
    - **sendmsg**
        - Owner: auth-cert-usr
        - User:: - - x
        - Group:: - - -
        - Other:: - - x
        - setuid bit set
    - **recvmsg**
        - Owner: auth-cert-usr
        - User:: - - x
        - Group:: - - -
        - Other:: - - x
        - setuid bit set
    - *outbound* folder
        - Owner: root

        - User:: r w x
        - Group:: - - -
        - Other:: r w x
    - *tmp* folder
        - Owner: auth-cert-usr
        - User:: r w x
        - Group:: - - -
        - Other:: - - -
    - *keypair* folder
        - Owner: get-cert-usr
        - User:: r w x
        - User:auth-cert-usr: r - x
        - Group:: - - -
        - Other:: - - -

Reasoning

- Server
    - The user can only start the request-handler executable which is the main server program that listens for client requests. Only the request-handler program can launch the other server programs
    - The user can't modify or delete any programs
    - The user can't access, modify, or delete any of the client passwords, client certs, or client mailboxes. Only users that the server programs run as are able to access the server resources, and these programs can only access the appropriate resources for the given program.
    - We design the server to emulate how the Apache web server works -- it must use root to execute chroot for sandboxing, but we shed privileges immediately after starting and only regain privileges when necessary (i.e. when executing the server programs with chroot and execl).
- Client
    - The client doesn't need to have access to their tls certificate. They can run the client programs, and users that the client programs run as can access the certificate as needed. By preventing the client from altering their certificate we can help to reduce the attack surface accessible to the clients.
    - The client is unable to modify or remove the client executables.

F. Security - Password Handling

- Whenever the users password is collected from the user it is collected using the getpass method to avoid displaying the password in plaintext on the users machine
- The password is hashed on the server side using a salt with Sha512. In this way plaintext passwords are not stored on the server.

G. Security - Certs

- server tls cert
    - Key Usage: critical, Digital Signature, Key Encipherment
- client tls/signing cert
    - Key Usage: critical, Digital Signature, Non Repudiation, Key Encipherment

I. Testing Plans

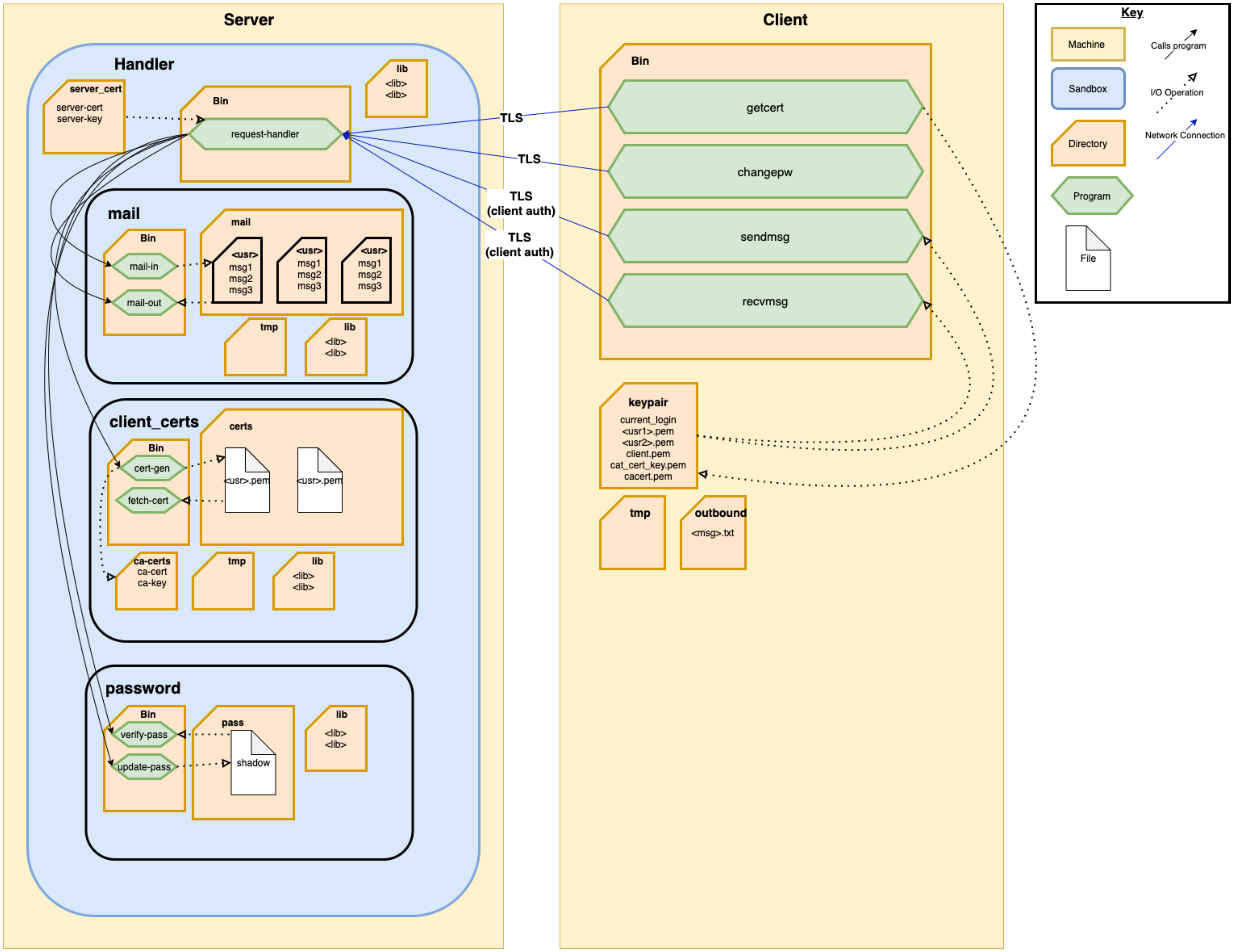
In this section, we detail some preliminary testing plans for each program. We will categorize these tests by client program workflow. At the bottom, we include the end-to-end tests we plan to also conduct, which'll include checking on both ends, server and client.

- **getcert**
    - Test #1: Valid Case
        - Client gives valid *username* and *password.*
        - Expected result: gets a valid client certificate in return.
    - Username Tests:
        - Test #2: Overflow attempt
            - Client gives a *username* that is too long.
            - Expected result: client gets an error message that denies the username and ends the session. No certificate is given.
        - Test #3: Invalid Username
            - Client gives a *username* that isn't on the system.
            - Expected result: client gets an error message that denies the username and ends the session. No certificate is given.
    - Password Tests:
        - Test #4: Wrong password, valid username
            - Client gives a valid *username* but an incorrect *password.*
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #5: Overflow attempt
            - Client gives a *password* that is too long.
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #6: Shell Injection
            - Client enters a password with shell injection
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
- **changepw**
    - Test #1: Valid Case
        - Client provides correct username password combination and a new password

        - Expected result: Password is successfully changed and the client receives a new certificate.
    - Test #2: Unread messages
        - Client tries to change password but their inbox has unread messages.
        - Expected result: client gets an error message notifying them they still have unread messages and the session is ended.
    - Username Tests:
        - Test #3: Overflow attempt
            - Client gives a *username* that is too long.
            - Expected result: client gets an error message that denies the username and ends the session. No certificate is given.
        - Test #4: Invalid Username
            - Client gives a *username* that isn't on the system.
            - Expected result: client gets an error message that denies the username and ends the session. No certificate is given.
    - Password Tests:
        - Test #5: Wrong password, valid username
            - Client gives a valid *username* but an incorrect *password.*
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #6: Overflow attempt on old password
            - Client gives a old *password* that is too long.
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #7: Overflow attempt on new password
            - Client gives a new *password* that is too long.
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #8: Shell Injection attempt on old password
            - Client enters a old password with shell injection
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
        - Test #9: Shell Injection attempt on new password
            - Client enters a new password with shell injection
            - Expected result: client gets an error message that denies the password and ends the session. No certificate is given.
- **sendmsg**
    - Test #1: Valid Case
        - Client provides a valid certificate, message, and at least 1 valid recipient.
        - Expected result: Message is successfully sent to the recipients.
    - Recipient List Tests
        - Test #2: Partially invalid recipient list
            - Client attempts to send a message to a recipient list that has an invalid recipient (but also one or more valid ones).

            - Expected result: no message is delivered to any inbox, and an error message is shown to the client.
        - Test #3: Invalid recipients (all invalid)
            - All recipients in the recipient list are invalid.
            - Expected result: no message is delivered to any inbox, and an error message is shown to the client.
        - Test #4: Overflow attempt on recipient username
            - Client gives a *username* that is too long in one of the recipient names listed.
            - Expected result: no message is delivered to any inbox, and an error message is shown to the client..
        - Test #5: Overflow attempt on recipient list- too many recipients
            - Client gives a *recipient list* containing too many recipients.
            - Expected result: no message is delivered to any inbox, and an error message is shown to the client.
    - Message Tests
        - Test #8: Overflow Attempt
            - Client attempts to overflow the program with a message that is too long.
            - Expected result: error message is shown to the client, and message is never sent.
            - Note: testing this locally provides the correct error message. Feel free to test this as well, the file limit size is 25mb.
        - Test #9: Message doesn't exist
            - Client attempts to send a message that doesnt exist.
            - Expected result: error message is shown to the client, and message is never sent.
        - NOTE: Shell injection is not an area of concern (or a valid case to throw an error) because message plaintext can and should be arbitrary (they are encrypted anyway).
- **recvmsg**
    - Test #1: Valid Case
        - Client provides a valid certificate.
        - Expected result: A pending message is printed successfully to stdout.
    - Test #2: No mail in inbox
        - Client provides a valid certificate.
        - Expected result: Client gets a message indicating there are no more messages in the mailbox currently for it to check.
- **end-to-end**
    - Test #1: Successfully send message and read (system test end-to-end)
        - Call getcert as one user. Call sendmsg to send a message to a different user. Call getcert as the user the message was sent to. Call recvmsg to display the just sent message.

        - Expected result: Get certificate of first user, successfully send message, get certificate of second user, successfully receive message.
    - NOTE: The user has no ability to switch out their certificate to another one or to delete it. Altering or removing the client certificate is not an area of concern.


Server
Handler
server_cert
server-cert
server-key
Bin
request-handler
lib
<lib>
<lib>
mail
Bin
mail-in
mail-out
mail
<usr>
msg1
msg2
msg3
<usr>
msg1
msg2
msg3
<usr>
msg1
msg2
msg3
tmp
lib
<lib>
<lib>
client_certs
Bin
cert-gen
fetch-cert
certs
<usr>.pem
<usr>.pem
ca-certs
ca-cert
ca-key
tmp
lib
<lib>
<lib>
password
Bin
verify-pass
update-pass
pass
shadow
lib
<lib>
<lib>
TLS
TLS
TLS
(client auth)
TLS
(client auth)
Client
Bin
getcert
changepw
sendmsg
recvmsg
keypair
current_login
<usr1>.pem
<usr2>.pem
client.pem
cat_cert_key.pem
cacert.pem
tmp
outbound
<msg>.txt
Key
Machine
Sandbox
Directory
Program
File
Calls program
I/O Operation
Network Connection