Advanced Computer Networks

Assignment 2 Documentation

Darryl Boggins, [redacted]

Table of Contents

Note: This project uses the python cryptography library: https://github.com/pyca/cryptography

Basic Running Guide

Change directory to the root of the project and run **python -m pip install -r requirements.txt**

After that, you can run the client with **python client.py**

You can run the server with **python -m flask –app server.py run**

Make sure you have the prerequisites for each (see **client.py** and **server.py**)

client.py

The entry point for the client. Exists to initialize the GUI, the rest of the work is done by other classes.

Note: Requires box developer token to run. This is stored as a plaintext file in auth/box_token.

Note 2: The client requires a copy of -server-public.pem, the servers public RSA key, to be present in the auth directory. See **server.py** for instructions on how to generate it.

The client has a few responsibilities:

- Non-group users don’t have access to anything but the filenames. They can view users, group users and log into box and see encrypted files, and upload their own unencrypted files, but they cannot decrypt the files.

- Group users can add or remove other users from the groups. They can also upload AES-256 encrypted files. Each file has a unique AES-256 key, and each group user with a public key has a unique public-key encrypted version of that key. Therefore, when a group user uploads an encrypted file, it is their responsibility to:

  - Create an AES key/iv and encrypt the file with it.
  - Collect the public keys of all group users from the server and verify the response with the server’s public key.
  - For each group user, encrypt the AES key with their public key (including the user themselves!) and distribute it to the server.

As such, when a group user adds a new group user:

  - Collect the public key of the new user
  - Inform the server that they’ve been added.
  - For each file, decrypt the AES key with their own private key, and re-encrypt it with the new user’s public key. Distribute that information to the server.

The server will then handle distributing keys to the appropriate client when a file is downloaded. Of course, it should be said that all important communications are verified with RSA; A SHA256 hash is created from the whole message, which is then signed with

by either's private key. The same hash is created on the receiving end, and it is checked against the signature.

It should be noted that this opens a potential vulnerability; with no key rotation, if a user is added to the group and somehow accesses the packet where they were added, they could re-send it if they're ever removed and assume the identity of the person who originally added them. One could consider implementing timestamps here, in some way, or rotating the keys regularly.

Creating a unique AES-256 key for each file and encrypting it for each user brings considerable overhead, but the reason I implemented it this way is because if a user is ever removed from the group, their machine will only ever had decrypted AES keys for files they already had access to, and none of the future files – these files I consider compromised already, as they could all be downloaded in plaintext. This does come with the limitation that this system can exponentially grow in size per file added; each one comes with a 256-byte overhead for **each user on the system**. For ten users with 50,000 encrypted files, this would be 128 megabytes plus the extra metadata needed to store each key and associate it to the user. If those 50000 files were about 100 bytes on average, they would only take up 5 megabytes on the disc – meaning their encryption data would be 2,560% larger.

server.py

The entry point for the server.

NOTE: Before running, you should set up the server's private key. This is done automatically by the script located in **scripts/make_server_key.py**. Remember the password, as this will be needed to start the server. This also generated -server-public.pem, the public key, which is needed for the client and stored in auth. The client should not have -server-private. You should also pre-authenticate a user with **scripts/make_preauth_user.py**, as this will allow you to log into a group account and start adding non-group users to it.

When starting up the server will:

- Check for any saved users/group users/encrypted keys and load them.

- Load pre-authenticated users (public keys present in the **preauth** directory)

- Load the server's private key (**auth/-server-private.pem**) and prompt the user for its password

- Start a flask app that listens for HTTP requests.

The server doesn't see any of the files on the box. It also doesn't receive any of the users private keys, nor does it receive any AES encryption keys directly. This is mostly handled by the users in the group; all the server does is store their public keys for distribution and verification of identity. It also stores AES keys encrypted by the users' public keys, which are encrypted and distributed by the users in the group. Therefore, the server can be breached and still provide no information to bad actors.

The server will sign each of its messages with its own private key and verify the users' messages against their public key, where needed.

As a last note, this server could theoretically be accessed from any machine in your house (or the world!) without much issue.

Function Documentation

Server

server.py

This is a flask application, so I'll be explaining the endpoints.

It should be noted that, when the server verifies with RSA, the POST data will look like {"json": **REQUEST HERE**, "signature": **SIGNATURE**}. As such, I won't be including this with every piece of documentation; it can be assumed if RSA verification is mentioned. When something is only available to group members, their signature will also be verified in a similar manner.

GET /getusers – RSA SIGNED

Returns a dictionary of usernames and their associated public keys.

GET /getgroup – RSA SIGNED

Ditto, but only for users in the group

GET /isuser/<name>

Returns 200 if the username is a registered user on the server, 404 if not.

GET /isgroup/<name>

Ditto, but for users being in the group.

POST /adduser

Registers a user in the server. Request is formatted {"user": username, "key": serialized PEM public key}. Only available for group members.

POST /addgroup – RSA SIGNED

Adds a user to the group. Request is formatted {"username": username of requestee, "username_to_add": username to be added}. Only available for group members.

POST /removegroup – RSA SIGNED

Similar, but removes from the group. {"username": username of requestee, "username_to_remove": username to be removed}. Only available for group members.

POST /getallkeys – RSA SIGNED

Returns a given users encrypted keys (if the requestee is in the group). {“username”: username of requestee, “username_to_fetch”: username to be fetched}. Returned json will have filenames as keys that contain the encrypted AES keys. Only available for group members.

POST /insertkeys – RSA SIGNED

Inserts one or more keys into the database. Request is formatted {“username”: username of requestee, “keys”: keys}, where keys is a dictionary of usernames to filenames to keys to store.

POST /getkey – RSA SIGNED

Gets a key for a specific file for a specific user. {“username”: username of user, “filename”: filename}. Only available for group members.

Client

login.py

This file creates a tkinter GUI for creating an account, registering with the server and creating/loading local .pem files. I won’t be going into detail about the UI, mostly just the encryption features.

self.on_button()

This is whats called when the login button is pressed. If no .pem exists locally, it’ll try to register one. If we have no local .pem but the username exists on the server, registration fails. Otherwise, we just log in. Password is needed to access the .pem file. Logging in is mostly local, there’s no session with the server and its just about accessing your .pem file. There are some limitations to what your username can contain, and what length your username/password must be.

main_ui.py

The main program UI. Has a number of methods that handle what the client should do.

self.refresh_all()

Refreshes user/group list from server and files from box. These responses are verified with the RSA public key (in the case of our local server).

self.on_download()

This function does a number of things. All responses from the server are verified.

- Retrieves our encrypted AES key from the server.
- Decodes it from Base64, decrypts it with our private key
- Downloads the selected file to a temp folder
- Decrypts the file with the AES key and stores it to the downloads folder.

self.on_upload()

All responses from the server are verified.

- Reads the file to be uploaded
- Encrypts it with AES and retrieves the key.
- Creates an encrypted key for the file for each user with their public key.
- Updates the server.

self.on_add_group()

All responses from the server are verified.

- Informs the server of the new user
- Retrieves all of our own encrypted AES keys, decrypts them, re-encrypts them with the new users public key and updates the server.

self.on_remove_group()

All responses from the server are verified.

- Removes the selected user from the group.

box.py

For maintaining connections with the box storage api at https://box.com.

self.update_auth(auth_code**)**

Updates developer token and opens a client.

self.upload_file(path, output_path**)**

Uploads a file (from path) to output_path on box.

self.download_file(path, file_id**)**

Downloads a file from file_id on box to path locally.

self.get_all_files(id**)**

Gets all files from a folder. ID is 0 by default, the root folder of box.

Utility

These are some functions I found useful on both the client and server, so I put them here.

globals.py

Contains some global values and utility functions.

serialize_public_key(public_key**)**

Converts a public key object to a string serialized PEM public key.

decrypt(message, private_key**)**

Decrypts a ciphertext message encrypted by a public key with its corresponding private_key, using the values I've been using for encryption/decryption

encrypt(message, public_key**)**

Ditto, but for encrypting with the public_key.

post_server(path, request**)** *and* **get_server(**path**)**

Helpful functions for making http requests to our local server. Path is the path we're requesting, request is a POST request as a dictionary.

sign_message(message, private_key**)**

Signs a message with the private key and returns the signature.

verify_message(message, signature, public_key**)**

Verifies a message with a signature and a public key.

build_response(response, private_key**)**

Builds a response in the format I've been using for this project.

The returned dictionary looks like {"json": [JSON STRING BEING SIGNED], "signature": [THE SIGNATURE]}

It takes in a dictionary, converts it to a json byte string and hashes it with SHA256. This hash is then signed, and that is inserted as the signature.

verify_response(response, public_key**)**

Verifies a response in the format I've been using for this project.

It hashes the json string and checks the signature against the public key we have. If it succeeds, it returns the json as a dictionary.

sha256_encrypt(plaintext_bytes**)**

Misleading name, but hashes plaintext_bytes with sha256.

aes_encrypt_file(input_bytes**)**

Encrypts bytes with AES-256. It uses AES-CTR. It generates a 32 byte key and 16 byte initialization vector. After the ciphertext is created, a tuple containing the joint key (key first, IV last) and the ciphertext is returned.

aes_decrypt_files(full_key, ciphertext**)**

Splits the key into its parts (0-31 bytes for key, 32-47 bytes for IV) and decrypts the ciphertext with it. Returns plaintext.