

INFO2222 Report

Group name: CC01_Team10

Name: Gordon Liu, Quynh Nhu Pham.

Summary of finished items according to the checking criteria:

- We managed to host our server on https.
- We managed to get the users' username, passwords, and store them securely in our sql database. For the password, we used sha256 to add on the security.
- We managed to display the friend list, and the place where the users can send the message and see their incoming message.
- We managed to generate RSA keypair, store the public key in the database, and store the private key into the cookies.
- We managed to do the encryption, but get some issues while storing it back to the database.
- We also figured out how to do the decryption, but because of having some issues with the database, we cannot fully test it.

Olivia:

- Certificate handling.
- Storing password (50%)
- Displaying friends list.
- Encryption.
- Report. (50%)
- Cookies generation attempts. (40%)

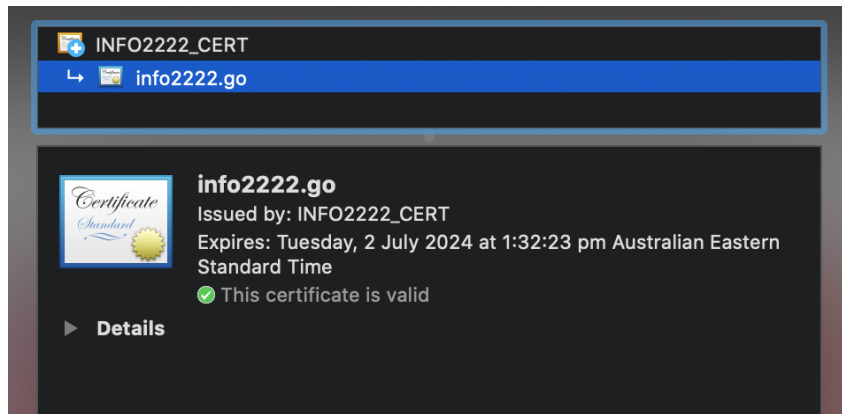
Gordon:

- Key generation in javascript
- Public key exporting and storage
- Password salting and storage (50%)
- Decryption
- Report writing (50%)
- Cookie generation (60%) for private keys

Body of report

1. Create a self-signed certificate.

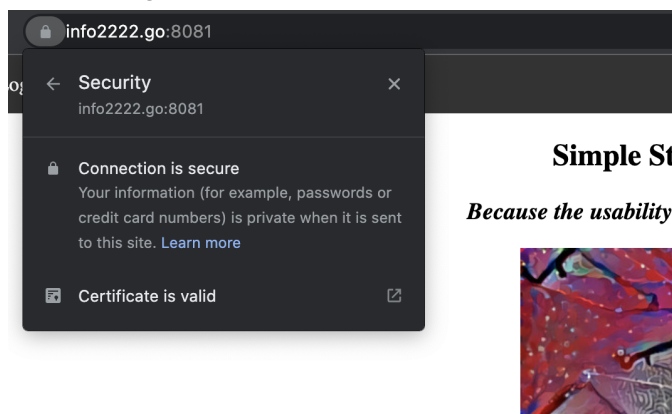
- We have our own authority named INFO2222_CERT, then we use this to sign our certificate (named info2222.go). This is the result:



- We modified the run.py a bit so that it will take in the certificate that we have created. It looks like this:

```
def run_server():
    """
    run_server
    Runs a bottle server
    """
    # run(host=host, port=port, debug=debug)
    # file_path = os.path.basename(os.path.dirname(__file__))
    key_path = './info2222.go.key'
    cert_path = './info2222.go.crt'
    config_path = './info2222.go.ext'
    run(host=host, port=port, debug=debug, server='gunicorn', keyfile=key_path, certfile=cert_path, config=config_path)
```

- When launching the server, it will say the connection is not secured at first. This is because 127.0.0.1 is not the name we registered for the certificate. Change 127.0.0.1 to info2222.go, our connection is now secured:



2. Safely store password in database

Put a route ('/register') in the controller.py. One function returns the form of the register. The other function is to get the users' input. Pass the inputs to the function called register_check in model. This is where we store the username and hashed password, and browser generated public key into the database (the key generation will be explained in the key generation and storage part down the end).

```
#-----  
@get('/register')  
def get_register_controller():  
    return model.register_form()  
  
# Attempt the register  
@post('/register')  
def post_register():  
    username = request.forms.get('username')  
    password = request.forms.get('password')  
    public_key = request.forms.get('PublicKey')  
    # print(public_key)  
    return model.register_check(username, password, public_key)
```

```
# Random Salt Generation and shuffle it twice  
salt = str(random.randint(0000, 9999)) + username  
rand_salt = ''.join(random.sample(salt, len(salt)))  
rand_salt = ''.join(random.sample(rand_salt, len(rand_salt)))  
  
# combine the salt with password  
salt_w_password = rand_salt + password  
# hash password  
hashed_password = hashlib.sha256(salt_w_password.encode()).hexdigest()  
  
sql_db.add_user(username, hashed_password, rand_salt, pub_key_db)  
sql_db.commit()
```

Our method of securing the password is to generate a hashed password with salting. Our rationale behind this is to ensure user passwords are safe from dictionary attacks. This is achieved by randomly generating a salt using our username and a random value from 0 to 9999. The salt string is then shuffled twice before combining it with the password to remove predictability. Finally, the salted password is hashed using a SHA-256 algorithm, which will then be stored in our sql database along with public key and usernames.

Now both the username and the hashed password will be stored into the database using add_user function. After all of this, this function will return the valid.html file, but if we have some errors, we will return invalid.

This is what it looks like when registered successfully.

Register with a username and password

Username:

Password:

This username-password combination is valid, welcome nhu!

3. Properly check whether password is correct

We have a function named `login_check` in the model file which is where we check if the user is the right person. We first get the salt based on the username by using the `get_salt` function which is in our sql file.

```
def get_salt(self, username):
    sql_query = """
        SELECT salt
        FROM Users
        WHERE username = '{username}'
    """

    sql_query = sql_query.format(username=username)
    print(username)
    self.cur.execute(sql_query)
    try:
        s = self.cur.fetchone()[0]
    except:
        return None
    # print(s[0])
    # return curs.fetchone()
    return s
```

If the salt is None, it means the username does not exist, returning the invalid page

```
login = True
sql_db = SQLiteDatabase('user1.db')

# getting the salt of this username
salt = sql_db.get_salt(username)
if salt == None:
    return page_view(["invalid", reason="username is not correct"])

# combine the salt with password
salt_w_password = salt + password
# hash pwd
hashed_password = hashlib.sha256(salt_w_password.encode()).hexdigest()
```

Otherwise, we get the salt, combine it with the to-check-password, use sha256 to hash it and then pass both the username and the hashed password to the function named

check_credentials. The function selects the password that is associated with the username where username equals the username that we passed in. Then we are going to compare whether the password the database returns is equal to the to-check-password we passed in. If it is, we return True, False otherwise.

```
if sql_db.check_credentials(username, hashed_password) == False:
    err_str = "Incorrect password"
    login = False

if login:
    friendlist = sql_db.get_user()
    return page_view("login_valid", name=username, list=friendlist)
else:
    return page_view("invalid", reason=err_str)
```

```
#-----
# Check login credentials (check password database match with username)
def check_credentials(self, username, password):
    sql_query = """
        SELECT username, password, salt
        FROM Users
        WHERE username = '{username}'
    """

    sql_query = sql_query.format(username=username)

    self.cur.execute(sql_query)
    to_compare = self.cur.fetchone()
    # print(to_compare)
    # print(password)

    if to_compare == None:
        return False
    elif to_compare[0] == username and to_compare[1] == password:
        return True
    else:
        return False
```

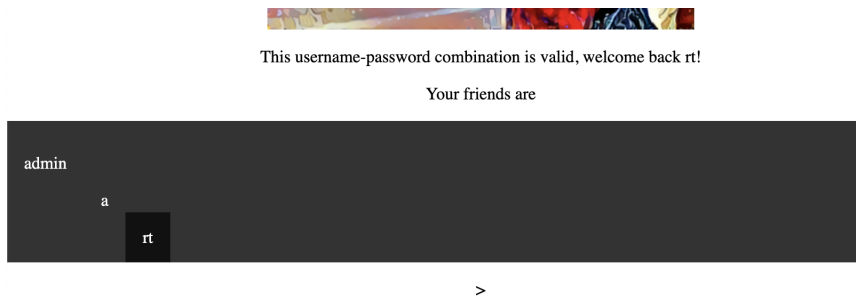
So if the login is not successful, we return invalid page view. Otherwise, the user will be directed to the page saying that they have logged in, and their friend list will be displayed. That is why we pass in another parameter called list=friendlist in the page_view when we return login_valid page. We get the friend list by retrieving all of the names in the database that we have so far. This is how we display the friend list

using the html:

```
<p>This username-password combination is valid, welcome back {{name}}!</p>
<p>Your friends are
% friends = list

<ul>
  % for friend in friends:
    <p><li><a href="/send/{{friend}}">{{friend}}</a></li><br>
    </p>
  % end
</ul>
</p>>
```

This is how it looks like in the website:



In the html file where we display the friendlist, we make each friend clickable. So when a user clicks onto the friend they want to send the message to which will lead them to the send message page.



4. Key Generation and Storage

Key generation was achieved within the html file "register.html" using a form and the crypto.subtle web API.

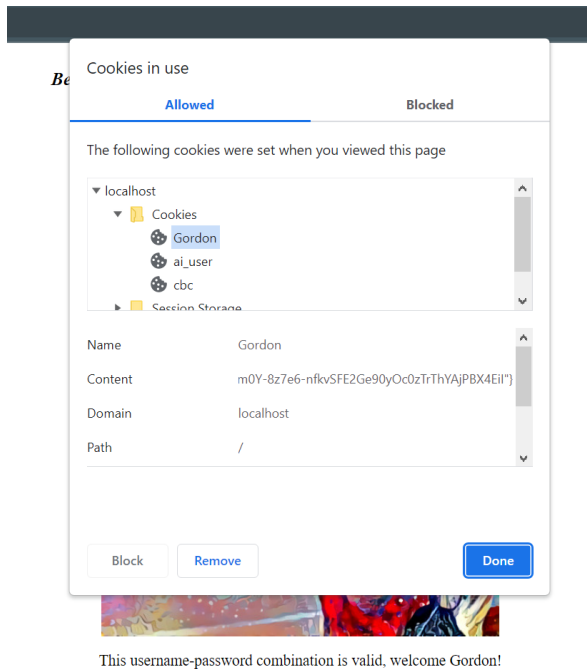
```
<p>
<form action="/register" id= "register" name="register" method="post" onsubmit="createAndExportKey();"
  Username: <input name="username" type="text" id="username"/>
</br>
  Password: <input name="password" type="password" />
</br>
  <input type="hidden" id="PublicKey" name="PublicKey" value="Error!!!" />
  <input value="Register" type="submit"/>
</form>
</p>
```

```

function createAndExportKey(){
  crypto.subtle.generateKey(
    {
      name: "RSA-OAEP",
      modulusLength: 2048,
      publicExponent: new Uint8Array([1, 0, 1]),
      hash: "SHA-256"
    },
    true,
    ["encrypt", "decrypt"]
  ).then(function(keyPair){
    // Export private key
    crypto.subtle.exportKey('jwk', keyPair.privateKey).then(function(jwk) {
      let jsonString = JSON.stringify(jwk);
      let name = document.register.username.value;
      document.cookie = name + "=" + jsonString;
      // alert("Your Cookie : " + document.cookie);
      // console.log(jsonString)
    });
    // Export public key
    crypto.subtle.exportKey("spki",keyPair.publicKey).then(function(result) {
      const exportedAsString = String.fromCharCode.apply(null, new Uint8Array(result))
      const exportedAsBase64 = window.btoa(exportedAsString);
      document.register.PublicKey.value = exportedAsBase64;
      // console.log(document.register.PublicKey.textContent)
      document.register.submit();
    });
  })
  // console.log(keyPair)
  return true;
}

```

When performing registration, the public key is stored as a hidden input within the registration form, with a default error value, then when the form is submitted, the form calls the function “createAndExportKey()”, which performs key generation using the “subtle.crypto” web API. We chose to generate a RSA keypair for increased security, using the RSA-OAEP algorithm and a modulus length of 2048. Upon key generation, the public and private keys are exported, with the private key being exported into a JSON web key and stored into the browser cookies, as shown below with a registered user. The public key is exported as a string and stored in the form element “PublicKey”, to be retrieved by the server and stored in the database upon user generation.



5. Messages Encryption

When a user clicks on the friend they want to send the messages to, the controller will graph the friend variable from `send/{{friend}}` route, and pass it on to the function like below.

```
# -----  
@get('/send/<username>')  
def get_sendmess_page(username):  
    return model.mess_form(username)
```

In `mess_form`, we are going to get the username's (friend a.k.a receiver) public key and pass it onto the `message_send` html file to encrypt the message.


```

#-----
# Send message
#-----
def mess_form(username):
    sql_db = SQLiteDatabase('user1.db')
    pub_k_string = sql_db.get_publickey(username)
    pub_k = read_public_key_as_PEM(pub_k_string)

    if pub_k is None:
        return page_view("invalid", reason="no such receiver")

    puo = pub_k.exportKey("PEM")

    return page_view("message_send", pubkey=puo, username=username)

```

This is message_send html file:

- Here we ask users to put in messages

```

<p>
<form action="/send" method="post">
|   <!-- Receiver: <input name="receiver" type="text" />
</br> -->
|   Please enter message below:
</br>
|   <input name="message" type="text" id="mess"/>
|
|   <button type="submit">Send</button>
|</form>
</p>

```

The script tag is where we perform the encryption (note: this script tag is in message_send html).

- encryptedBase64 will be where our encrypted message is.
- We assign the public key to the pemEncodedKey variable. We will need this to import the public key to encrypt the message later.
- Assign encryptedBase64 to id encrypted so that we can retrieve it in the controller later.

```

<script id="encrypted" type="module">
    // let importedPub;
    let encryptedBase64;
    const pemEncodedKey = `{{pubkey}}`;

    importPublicKeyAndEncrypt();
    document.getElementById('encrypted').innerHTML = encryptedBase64;

```

- importPublicKeyAndEncrypt function pulls the message input by id “mess” from the form above. Import the key and get the key and the message to put into the encrypt function. Lastly, convert it into base64 which is a string data type so that we can store it in a sql database.

```
async function importPublicKeyAndEncrypt() {
  const messageBox = document.getElementById("mess");
  let message = messageBox.value;
  let enc = new TextEncoder();
  var mess = enc.encode(message);

  try {
    const pub = await importPublicKey(pemEncodedKey);
    const encrypted = await encryptRSA(pub, mess);
    encryptedBase64 = window.btoa(ab2str(encrypted));

    console.log(encryptedBase64.replace(/(.{64})/g, "$1\n"));
  } catch(error) {
    console.log(error);
  }
}
```

- Import the key to spki form from the pemEncodedKey.

```
async function importPublicKey(spkiPem) {
  return await window.crypto.subtle.importKey(
    "spki",
    getSpkiDer(spkiPem),
    {
      name: "RSA-OAEP",
      hash: "SHA-256",
    },
    true,
    ["encrypt"]
  );
}
```

- Use the key and the message to encrypt.

```

async function encryptRSA(key, plaintext) {
    let encrypted = await window.crypto.subtle.encrypt(
        {
            name: "RSA-OAEP"
        },
        key,
        plaintext
    );
    return encrypted;
}

```

- After getting the encryption done, which is when the user presses the send button, the controller will retrieve the encrypted message and the username, and pass it on to the send_mess model.

```

# send message
@post('/send/<username>')
def post_message(username):
    # receiver = request.forms.get('receiver')
    # message = request.forms.get('message')
    encrypted = request.get('encrypted')
    # print(encrypted)
    return model.send_mess(username, encrypted)

```

- send_mess will store the encrypted message in the database and return a html file saying the user has sent the message successfully.

```

def send_mess(receiver, message):
    sql_db = SQLiteDatabase('user1.db')
    sql_db.add_mess([receiver, message])
    return page_view("send_result", name=receiver)

```

```

def add_mess(self, name, mess):
    sql_query = """
        INSERT INTO Encrypted
        VALUES('{receiver}', '{ciphermessages}'))
    """

    # print(name)
    # print(mess)

    sql_cmd = sql_cmd.format(receiver=name, ciphermessages=mess)
    self.cur.execute(sql_cmd)
    self.commit()

```



You have sent message to nhu successfully!!

When I print out the encrypted message to test if it is working, we have it in the console, so it means we can encrypt the message successfully:

```
html
Console What's New Issues
[play] [stop] top [eye] Filter Default levels 1 Issue: 1 [gear]
EflnWE/7jexUNrapFQ2nvKMubdAhCMzCL4D0avQnzUman8T5y/i1hhUGPJ0q4gwD
jFgoAkvIj8RWQito/ZsvEkUR1S+1CE0KNlhJAKUt/D+YJdtorrLZY1GN1KgJfhYw
2lHJAwGS41JhdkhqMzGFrK8xuwJeshokd9mg0ZnweDmoqNqTvcWn3ef4AL4NMizn
h80okVP9rA3PcTBg5Xwi3zMbcDbc1Qprnp0nek1h+KF16XorHb3cdj1KpeJRzsIs
aV/M/epQwqznnGr2vRRn1J0sfrMLxGdNm92bhNT0X6UXQkZHLIXZXln0Yp80e97F
L0E3SwR0FyQ7CGo+ldae1w== send:66
```

5. Decryption

While decryption is not completed due to difficulties, the following paragraph would outline the main steps we would go through to decrypt our incoming messages.

We would achieve decryption in the “incoming.html” page using the “subtle.crypto” web API. Firstly, we would retrieve the private key from cookies and parse it back into a JSON web key format to be used for decryption. We would then convert the selected ciphertext back into an Array Byte to prepare it to be converted back into plaintext. Then the private key is used to decrypt the ciphertext displayed in the message list back into plaintext using the RSA-OAEP algorithm, and the plaintext is displayed back on screen, replacing the ciphertext’s position in the document.

Process undertaken by group

- We used [Github](#) for code maintenance, and revision control.
- We used OpenSSL to generate certificates.
- We used the Web API CryptoSubtle for key generation, encryption and decryption, due to its use in javascript, which was ideal for user end-to-end encryption.
- We used the python module hashlib for sha256 symmetric encryption for password.
- We used SQLITE3 database to store users’ information and encrypted messages.
- We used the Simple Template Bottle to render our web server.
- We used gunicorn to host our server.

Project limitation discussion

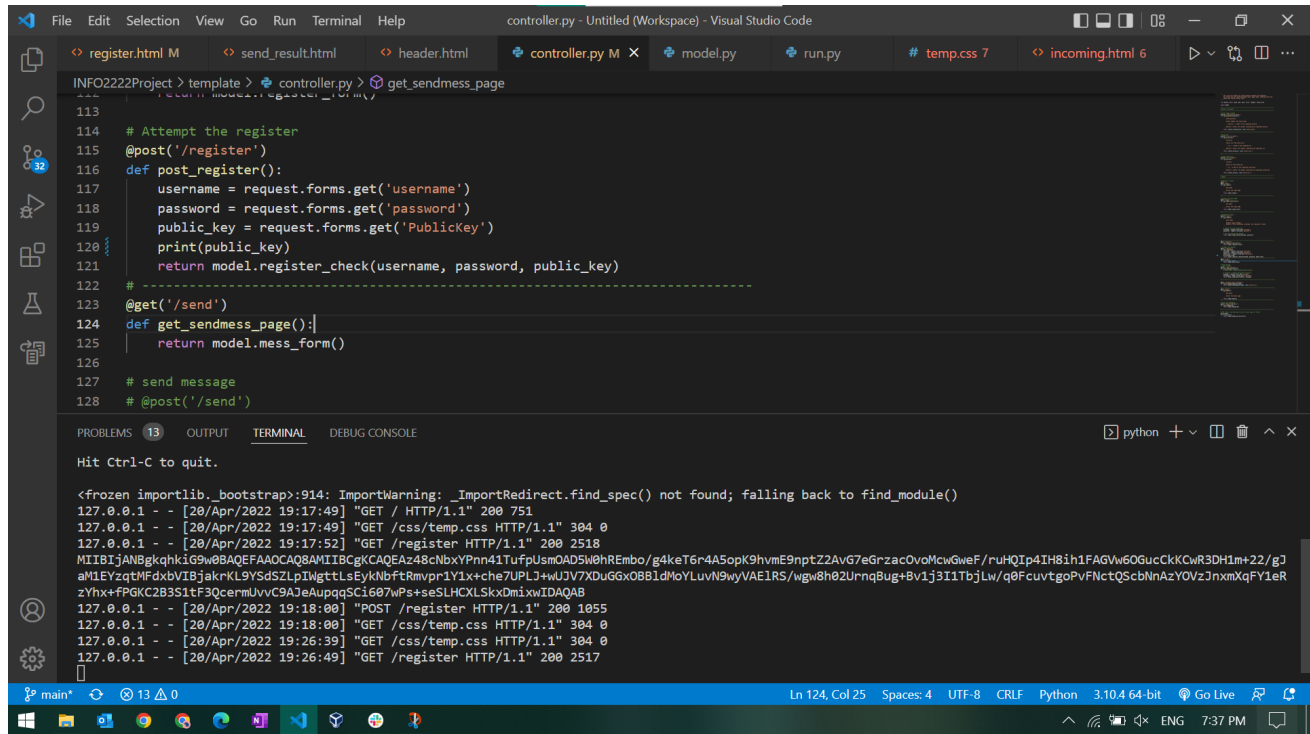
- Since both of members does not have knowledge regarding to web such as html, css, javascript before, it takes a bit of time in the beginning for us to get used to the template.
- Also this is the first time we use sql as well, so take time to figure how to store and retrieve

data from sql.

- Also because this is a big project, sometimes we modify lots of files. This sometimes causes some issues while using git to push and pull code.

Appendix

Running key generation in localhost. There appears to be some issues running key generation in the server, where subtle crypto does not run even on the secure website.



```
INFO2222Project > template > controller.py > get_sendmess_page
113
114 # Attempt the register
115 @post('/register')
116 def post_register():
117     username = request.forms.get('username')
118     password = request.forms.get('password')
119     public_key = request.forms.get('PublicKey')
120     print(public_key)
121     return model.register_check(username, password, public_key)
122
123 @get('/send')
124 def get_sendmess_page():
125     return model.mess_form()
126
127 # send message
128 @post('/send')
```

```
Hit Ctrl-C to quit.
<frozen importlib._bootstrap>:914: ImportWarning: _ImportRedirect.find_spec() not found; falling back to find_module()
127.0.0.1 - - [20/Apr/2022 19:17:49] "GET / HTTP/1.1" 200 751
127.0.0.1 - - [20/Apr/2022 19:17:49] "GET /css/temp.css HTTP/1.1" 304 0
127.0.0.1 - - [20/Apr/2022 19:17:52] "GET /register HTTP/1.1" 200 2518
MIIEBjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAz48cNbxYPnn41TufpUsmOAd5W0hREmbo/g4keT6r4A5opK9hvmE9nptZ2AvG7eGrzacOvoMcvGweF/ruHQIp4IH8ih1FAGVw6OGucCKKwR3DH1m+22/gJ
aM1EYzqtMfDxbVIBjakrKL9YsdsZLpIwgttLsEyknBftRmvpr1Y1x+che7UPLJ+wUJV7XDUGx0B8ldMoYLuVn9wyVAE1RS/wgw8h02UrnqBug+Bv1j3II1bJLw/q0FcvutgoPvFNctQScbNnAzYOVzJnxmXqFY1eR
zyhx+fPGKC2B3S1tF3QcermUvvc9AJeAupqqSCi607wPs+seSLHCXLSkxDmixwIDAQAB
127.0.0.1 - - [20/Apr/2022 19:18:00] "POST /register HTTP/1.1" 200 1055
127.0.0.1 - - [20/Apr/2022 19:18:00] "GET /css/temp.css HTTP/1.1" 304 0
127.0.0.1 - - [20/Apr/2022 19:26:39] "GET /css/temp.css HTTP/1.1" 304 0
127.0.0.1 - - [20/Apr/2022 19:26:49] "GET /register HTTP/1.1" 200 2517
```

The decryption wasn't fully tested, due to issues with passing variables into the webpage, but the code is completed and can be found in the repository link or in the incoming.html:

https://github.com/Gordon-4389/INFO2222Project/blob/main/incoming_decryption.html