COMP3334 Project Demonstration - Group 19

Presented by Zeng Tianyi

Zeng Tianyi, Wang Ruijie, Zhu Jin Shun, Liu Yuyang

Deparment of Computing, The Hong Kong Polytechnic University

THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

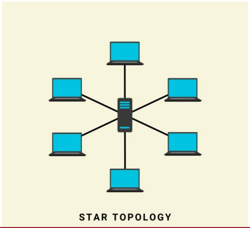
Introduction

- Online storage is frequently applied in our daily life. It allows users to upload files to a server and retrieve them when needed.
- However, files may contain sensitive information. If the system is not secure, serious consequences could occur, including but not limited to the leakage of sensitive information.
- Therefore, it is necessary to ensure the online storage system is immune from common threats. Our team has designed and developed a command-line based secure system for file storage and retrieval.

Overview of our system's architecture

We adapt the popular client-server software framework with the star topology.

- The server-side application is the only centralized node that is responsible for storing data and processing client requests.
- The client-side application implements user interface, encryption, and decryption.
- The *functions.py* program serves as a local package to provide detailed function implementations and allow easy invocation by the system.



Section 1: An overview of the system’s functionality

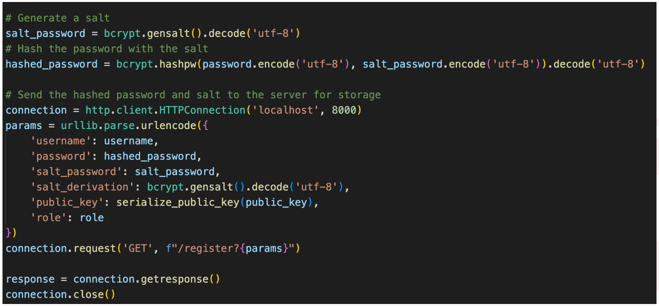
- Our system covers all core functions as required, including user registration and login, file uploading and downloading, access control with file sharing, log auditing, and general security protection (i.e., against SQL injection and file name attack).
- We also implement an extended function by supporting multi-factor authentication. We employ email verification code as a kind of OTP to support user login by email.

Section 2: Elaboration on **User Management**, part 1 of core functionalities

1.a: Our system supports secure user registration.

- During registration, username and password will be required. The input username will be examined to avoid name conflicts.
- We employ a password hashing mechanism with the **bcrypt** library and transmit the hashed password and the corresponding random **salt** to the server-side application to store in the database.
- The password is hashed at the client-side application, preventing the password leakage during client-server transmission.
- We also require clients to specify their roles (i.e., "user" or "admin") during registration for the privilege assignment. Only admins may export the audit logs for investigation purposes.

Section 2: Elaboration on **User Management**, part 1 of core functionalities



```
# Generate a salt
salt_password = bcrypt.gensalt().decode('utf-8')
# Hash the password with the salt
hashed_password = bcrypt.hashpw(password.encode('utf-8'), salt_password.encode('utf-8')).decode('utf-8')

# Send the hashed password and salt to the server for storage
connection = http.client.HTTPConnection('localhost', 8000)
params = urllib.parse.urlencode({
    'username': username,
    'password': hashed_password,
    'salt_password': salt_password,
    'salt_derivation': bcrypt.gensalt().decode('utf-8'),
    'public_key': serialize_public_key(public_key),
    'role': role
})
connection.request('GET', f"/register?{params}")

response = connection.getresponse()
connection.close()
```

Figure 1: This figure shows our password hashing implementation. Salt is applied to enhance the hashing strength

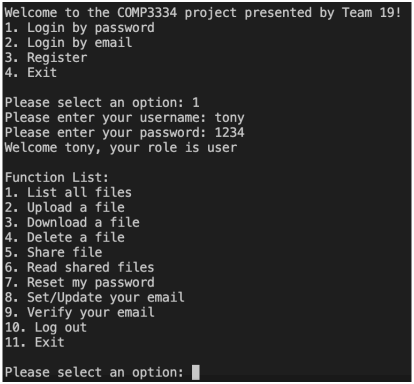
Section 2: Elaboration on **User Management**, part 1 of core functionalities

1.b: Our system supports user login with 2 methods.

- A user can log in by password. The program will read the input username and get the corresponding salt from the database. Then the input password will be hashed in the same way and sent to the server for verification.
- A user may also log in by email if he has set and verified his email in the system. If satisfied, the system will generate an OTP and send it to the user's email. By requesting for the user's input of OTP, the system can determine whether to authenticate the user and allow log in. The detail of this function will be introduced later in the extended functionality part.

Section 2: Elaboration on **User Management**, part 1 of core functionalities

1.b: Our system supports user login with 2 methods.

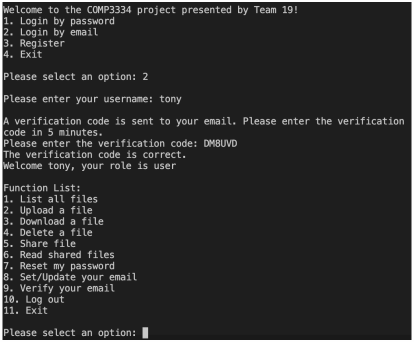


```
Welcome to the COMP3334 project presented by Team 19!
1. Login by password
2. Login by email
3. Register
4. Exit

Please select an option: 1
Please enter your username: tony
Please enter your password: 1234
Welcome tony, your role is user

Function List:
1. List all files
2. Upload a file
3. Download a file
4. Delete a file
5. Share file
6. Read shared files
7. Reset my password
8. Set/Update your email
9. Verify your email
10. Log out
11. Exit

Please select an option:
```

Figure 2: Method 1: log in by password

Section 2: Elaboration on **User Management**, part 1 of core functionalities

1.b: Our system supports user login with 2 methods.



```
Welcome to the COMP3334 project presented by Team 19!
1. Login by password
2. Login by email
3. Register
4. Exit

Please select an option: 2

Please enter your username: tony

A verification code is sent to your email. Please enter the verification
code in 5 minutes.
Please enter the verification code: DM8UVD
The verification code is correct.
Welcome tony, your role is user

Function List:
1. List all files
2. Upload a file
3. Download a file
4. Delete a file
5. Share file
6. Read shared files
7. Reset my password
8. Set/Update your email
9. Verify your email
10. Log out
11. Exit

Please select an option: █
```

Figure 3: Method 2: log in by email

Section 2: Elaboration on **User Management**, part 1 of core functionalities

1.c: Our system allows the user to reset his password.

- We require the user to input a different password. The program will retrieve the user's salt and current hashed password from the database. Then, the user's input will be hashed with the salt and compared with the stored hash value. If they are identical, the action is declined.
- If the hash comparison is successful, the newly-hashed password and its salt will be sent to the database to update the record.

Section 2: Elaboration on **Data Encryption**, part 2 of core functionalities

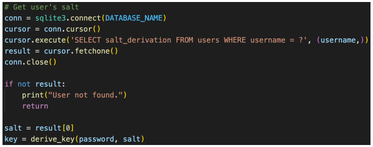
2.a: Our system supports secure file upload.

- The program asks for user input of the path of the file to be uploaded and conducts file name validation (refer to 5.a).
- Then the user's salt and hashed password will be retrieved from the database, preparing for a later password verification. This is to ensure it is the claimed user to upload the file.
- An AES key will be derived from the password (in plaintext, retrieved from user input in the verification phase) and the salt to encrypt the file to be uploaded.
- Since the key is derived from the password, there is no need to store the key, preventing the risk of leakage.
- The file encryption happens at client side, ensuring the transmission to server and database is secure.

Section 2: Elaboration on **Data Encryption**, part 2 of core functionalities

2.b: Our system supports secure file download.

- To download, the program asks the user to input the name of the file and his password.
- The password and the user's corresponding salt are used to derive the key to decrypt the file.
- The file is first downloaded in its encrypted form and then decrypted at the client side. There's no risk in transmission.

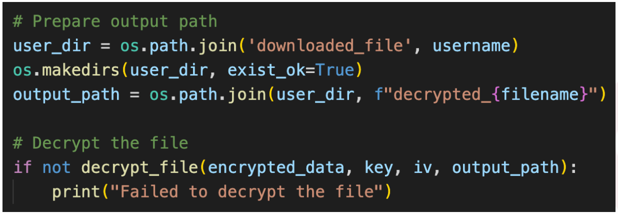


```python
# Get user's salt
conn = sqlite3.connect(DATABASE_NAME)
cursor = conn.cursor()
cursor.execute('SELECT salt_derivation FROM users WHERE username = ?', (username,))
result = cursor.fetchone()
conn.close()

if not result:
    print("User not found.")
    return

salt = result[0]
key = derive_key(password, salt)
```

Figure 4: The key to decrypt the file, derived from password and salt

Section 2: Elaboration on **Data Encryption**, part 2 of core functionalities

2.b: Our system supports secure file download.



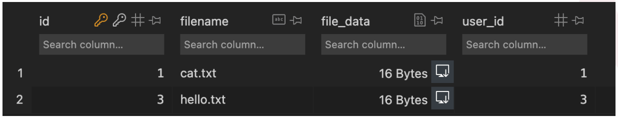
```python
# Prepare output path
user_dir = os.path.join('downloaded_file', username)
os.makedirs(user_dir, exist_ok=True)
output_path = os.path.join(user_dir, f"decrypted_{filename}")

# Decrypt the file
if not decrypt_file(encrypted_data, key, iv, output_path):
    print("Failed to decrypt the file")
```

Figure 5: The program will export the downloaded and decrypted file to a specific folder in the current working directory

Section 3: Elaboration on **Access Control**, part 3 of core functionalities

3.a: A user can only add/edit/delete its own files.

- We design a mechanism to realize this requirement. A user must enter his password when uploading or downloading a file.
- Besides, files stored in the database are associated with the id of the users who upload them. Other users don't have access.



	id	filename	file_data	user_id
	Search column...	Search column...	Search column...	Search column...
1	1	cat.txt	16 Bytes	1
2	3	hello.txt	16 Bytes	3

Figure 6: The schema of the file table that records the id of the file owner to ensure access control

Section 3: Elaboration on **Access Control**, part 3 of core functionalities

3.b: Our system supports file sharing and reading shared files.

- We design the file sharing workflow as follows: the program will ask for inputs of the file name, the target user name, and the owner's password.
- Then, the program retrieves the target user's RSA public key and the file owner's salt. The real key to encrypt the shared file is derived from the owner's password and salt, and this real key is further encrypted by the target user's public key.
- Therefore, only the target user can use his secret key to decrypt the real key and then use the real key to decrypt the shared file. Each user's private key is generated during registration, and it's encrypted with the user's password and an IV.

Section 3: Elaboration on **Access Control**, part 3 of core functionalities

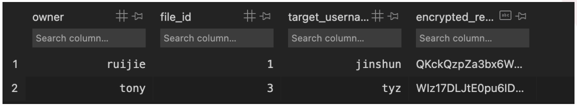
3.b: Our system supports file sharing and reading shared files.

- When the target user wants to read a shared file, he needs to input his password to decrypt his private key first. Afterwards, the program will use his private key to decrypt the encrypted real key.
- With the real key, the shared file can be decrypted. The plaintext shared file will be exported to a specific path for the target user to read.
- An illustration of this mechanism: File <–(encrypt) Real key <–(encrypt) Target user's public key; Target user's private key –>(decrypt) encrypted real key; Target user's private key <–(encrypt) Target user's password and IV

Section 3: Elaboration on **Access Control**, part 3 of core functionalities

3.c: Our system protects files from unauthorized access.

- As mentioned earlier, each uploaded file is associated with its owner id, and each shared file is associated with its target user id.
- There is no possibility that an unauthorized user has access to others' files that have no relation to him.



	owner	file_id	target_userna...	encrypted_re...
	Search column...	Search column...	Search column...	Search column...
1	ruijie	1	jinshun	QKckQzpZa3bx6W...
2	tony	3	tyz	WIz17DLJtE0pu6ID...

Figure 7: The schema of the shared file table that builds a connection between the file and the target user

Section 4: Elaboration on **Log Auditing**, part 4 of core functionalities

4.a & **4.b**: Our system records all critical operations to provide non-repudiation. Only admins can access the log file

- We implement the logging function by designing a table in the database and storing all log records in it.
- The attributes of a log include its unique id, timestamp, action type, user name, details (e.g., user A uploads a file X), and status.
- Actions that will be recorded: registration, login, logout, password resetting, uploading, downloading, deleting, file sharing, reading shared files, setting/updating email, and verifying email.

Section 4: Elaboration on **Log Auditing**, part 4 of core functionalities

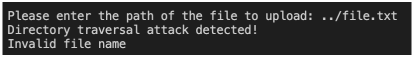
4.a & **4.b**: Our system records all critical operations to provide non-repudiation. Only admins can access the log file

- Our design ensures all logs can only be retrieved from the database, and it is not directly accessible to clients.
- Only admins can export the log file from the database. To distinguish user roles, we require clients to specify their role (i.e., user or admin) during registration.

Section 5: Elaboration on **General Security Protection**, part 5 of core functionalities

5.a: Our system can defend file name attacks.

- We set the rules of evaluating the validity of a file name. We mainly avoid 3 situations: 1) directory travesal attack (e.g., ../file.txt can be used to access *file.txt* in the parent folder), 2) retained device names (e.g., CON, PRN, and AUX are reserved names in Windows), and 3) length overflow (i.e., file name cannot exceed 255 bytes).



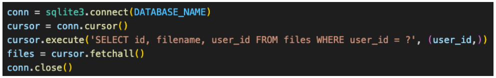
```
Please enter the path of the file to upload: ../file.txt
Directory traversal attack detected!
Invalid file name
```

Figure 8: An example of invalid file name:../file.txt, which is used in directory traversal attack.

Section 5: Elaboration on **General Security Protection**, part 5 of core functionalities

5.b: Our system can withstand SQL injections.

- All database operations use parameterized queries with placeholders (i.e., ?). With this method, the database driver will automatically encode user inputs.
- This ensures user inputs are treated as data rather than executable SQL codes to avoid the execution of any unintended database operation.



```
conn = sqlite3.connect(DATABASE_NAME)
cursor = conn.cursor()
cursor.execute('SELECT id, filename, user_id FROM files WHERE user_id = ?', (user_id,))
files = cursor.fetchall()
conn.close()
```

Figure 9: An example of parameterized database operations

Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities

Besides all core functions, we also implements email verification code as an OTP mechanism.

- To use this function, a user needs to set his email after login. If he already set before, he may update the email.
- After his email is stored in the database, he needs to verify his email. Our system will send an OTP to that address, and the user needs to input that OTP to verify his email.
- When the email is verified, the user may log in by email in the future with the same procedure in the email verification mentioned above. The example is shown in Figure 3.

Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities

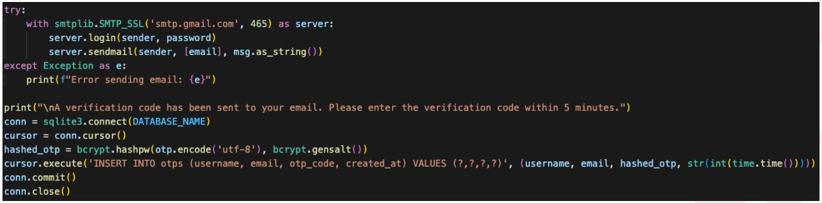
Besides all core functions, we also implements email verification code as an OTP mechanism.

- The key behind this function is the generation, storage, and verification of OTPs.
- Our 6-digit OTP is generated by a random combination of digits and uppercase letters. There is an OTP table in the database to temporarily store OTPs. When a user is doing OTP-related operations, the generated OTP will be hashed and stored with the corresponding username and his email.

Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities

- The generated OTP is sent to the user's email via SMTP protocol, and it's valid for 5 minutes due to safety concerns. When the user inputs the OTP he receives, the program will first hash his input and then compare with the hashed OTP in the database. If verified, the system will delete that OTP record in the table to avoid reuse.

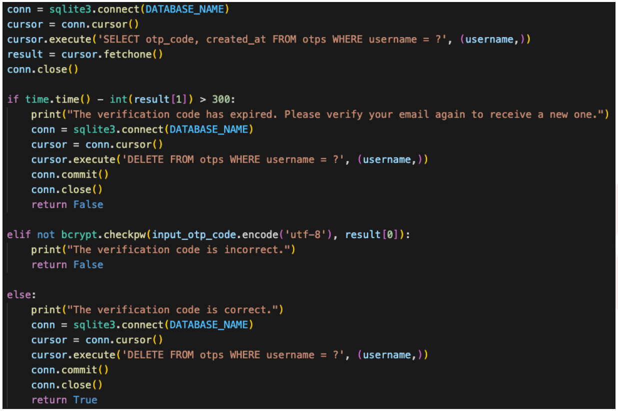
Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities



```
try:
    with smtplib.SMTP_SSL('smtp.gmail.com', 465) as server:
        server.login(sender, password)
        server.sendmail(sender, [email], msg.as_string())
except Exception as e:
    print(f"Error sending email: {e}")

print("\nA verification code has been sent to your email. Please enter the verification code within 5 minutes.")
conn = sqlite3.connect(DATABASE_NAME)
cursor = conn.cursor()
hashed_otp = bcrypt.hashpw(otp.encode('utf-8'), bcrypt.gensalt())
cursor.execute('INSERT INTO otps (username, email, otp_code, created_at) VALUES (?,?,?,?)', (username, email, hashed_otp, str(int(time.time()))))
conn.commit()
conn.close()
```

Figure 10: Sending OTP to email and store the relevant record in the database

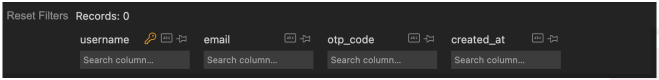
Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities



```
conn = sqlite3.connect(DATABASE_NAME)
cursor = conn.cursor()
cursor.execute('SELECT otp_code, created_at FROM otps WHERE username = ?', (username,))
result = cursor.fetchone()
conn.close()

if time.time() - int(result[1]) > 300:
    print("The verification code has expired. Please verify your email again to receive a new one.")
    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()
    cursor.execute('DELETE FROM otps WHERE username = ?', (username,))
    conn.commit()
    conn.close()
    return False

elif not bcrypt.checkpw(input_otp_code.encode('utf-8'), result[0]):
    print("The verification code is incorrect.")
    return False

else:
    print("The verification code is correct.")
    conn = sqlite3.connect(DATABASE_NAME)
    cursor = conn.cursor()
    cursor.execute('DELETE FROM otps WHERE username = ?', (username,))
    conn.commit()
    conn.close()
    return True
```

Figure 11: Examine the OTP input by the user. Verification will fail if the code is expired or incorrect.

Section 6: Elaboration on **Multi-Factor Authentication**, part 1 of extended functionalities



Reset Filters Records: 0

username	email	otp_code	created_at
Search column...	Search column...	Search column...	Search column...

Figure 12: The schema of the OTP table. OTP code is the hashed OTP that ensures no plaintext OTP is stored in the database.

Conclusion

- Our online storage system has multiple functions, such as uploading, downloading, and file sharing.
- Information security is achieved generally by encryption and cryptographic hashing. There will be no plaintext sensitive information in the server and during transmission.
- Other threats are also considered with corresponding measures. The system can withstand SQL injections and file name attacks, further enhancing its security feature.

Demo

Now, please watch our demo for using our system. Thank you.