

## INFO2222 Assignment Part 1 Report

**Group name:** CC01\_Team4  
**Names:** Cooper Mills, Owen Booth

### \*IMPORTANT NOTE\*

We have set up two sample users who are friends for you to use in the testing:

Username	Password
owen	owen
cooper	cooper

Registration is fully functional so in the case you would like to register new users and add friends in the sql database, refer to sql.py line 87.

### Summary:

#### List of finished items in project criteria:

- Passwords can be properly stored on the server (Examine Criteria 1)
- Server's certificate check is enabled (Examine Criteria 2)
- Passwords can be securely transmitted to the server (Examine Criteria 3)
- Credential check, including checking for correct password (Examine Criteria 4)
- Secure transmission of messages between two users (Examine Criteria 5)

#### Individual Contributions:

##### Cooper Mills:

- Secure storage of passwords on the server
- Server certificate check
- Secure transmission of passwords to the server
- Secure transmission of messages between two users

##### Owen Booth:

- Implement credentials check
- Writing report
- Secure transmission of messages between two users

### Body of report:

#### Tools and techniques used for each Examine Criteria:

## 1. Storing passwords properly on the server

To properly store passwords on the server we used the module hashlib's key derivation function with the following signature: pbkdf2\_hmac(hash\_name, password, salt, iterations).

The hash digest algorithm we have opted for is 'sha-256'. This is applied to a 'utf-8' encoded password set by a user upon registration.

Passwords are also salted with a salt of 32 randomly generated bytes, using the os module. Iterations have been set to the suggested 100000 as we are using 'sha-256'. As this is a development site with low traffic, 100000 iterations is tolerable for our application.

Finally the hashed and salted password is stored in hexadecimal format in the database using the .hex() function. Evidence of the proper storage of passwords on the web server can be found in Appendix [1.1](#) and [1.2](#))

## 2. Server's certificate check is enabled

As we are using macOS, we followed the prompts in the KeyChain App, a key management utility, to create our own certificate authority. Firstly, we generated a private key and root certificate using the OpenSSL command-line tool and the following commands:

```
openssl genrsa -des3 -out cert.key 2048 (To generate the private key)
```

```
openssl req -x509 -new -nodes -key cert.key -sha256 -days 1825 -out cert.pem (To generate the root certificate).
```

Then we imported the root certificate to the KeyChain App and selected 'Always Trust when using this certificate'.

From here, we were now able to create a self-signed certificate for our web server. We created a private key and certificate signing request in OpenSSL with the following commands:

```
openssl genrsa -out server.key 2048 (To generate the private key for the site)
```

```
openssl req -new -key server.key -out server.csr (To create the certificate signing request).
```

We then created a configuration file server.ext in which we were able to specify all the DNS Names, i.e., pages on our site, that would be protected by this certificate.

Returning to OpenSSL we then created the certificate using our certificate authority's private key and root certificate as well as the certificate signing request and configuration file we previously created using the following command:

```
openssl x509 -req -in server.csr -CA cert.pem -CAkey cert.key -CAcreateserial -out\ server.crt -days 825 -sha256 -extfile server.ext
```

We chose to use the gunicorn package as our Web Server Gateway Interface as it supports TLS. To configure it to listen on https, we invoked 'gunicorn' in our bottle run function and specified the 'certfile' as the certificate created for the web server in the steps above, converted

to .pem format, and 'keyfile' as the private key generated in the steps above (as can be seen in Appendix [2.1](#)).

### 3. Passwords can be securely transmitted to the server

As demonstrated above, the server's certificate check has been established and TLS encryption is in place.

During the TLS 1.2 (1.3 is not yet supported by gunicorn) handshake, the user's web browser contributes the pre-master secret, encrypts it with the server's public key that is contained within the certificate and sends this to the server. A MAC is attached to this information to ensure integrity and authentication. The server then decrypts this pre-master secret with its private key to compute the session key. This session key ensures encrypted communication between our server and a client.

This means data breaches and any attempt of a 'man in the middle' attack between a user's browser and the web server is highly unlikely. Secure connection to the server is shown in Appendix [3.1](#).

### 4. Credential check, including checking for correct password

When a user attempts to log in, a check function is called to verify their credentials. The hashlib pbkdf2\_hmac function, explained in project criteria 1, is used to compute and subsequently verify the password entered matches the password stored in the database for the corresponding username entered.

Additionally mentioned in project criteria 1 was the use of password salting. As the salt is stored on the browser, this defends against offline pre-computational attacks as the attacker doesn't have access to the 32 byte random string that salts the passwords.

### 5. Secure transmission of messages between two users

The API SubtleCrypto is used when a user logs in to create an RSA key pair for the user. The private key is stored client side in indexeddb (Appendix [5.1](#)), and the public key is stored server side (Appendix [1.2](#)). When a user sends a message they first input the receiver, the server sending them the required public key, the message is then inputted by the user, encrypted and sent. When any user navigates to the "/chat" domain the server will send them any encrypted messages they may have. The client will then access their stored private key and decrypt the messages (Appendix [5.2](#)).

However, currently, the messaging between two friends is limited based on some confusion with asynchronous functions, wait calls, server & client communication, etc. Therefore, the sending of messages currently only works once and only one way.

This ensures confidentiality as only the sender and intended recipient know the contents of the message. Whilst integrity is somewhat implied given the circumstances (only one person has access to a given account), in reality we would have liked to implement HMAC to ensure message integrity, given we had more time and guidance.

## Appendix:

### 1.1

```
2 import os
3 import hashlib
4
5 salt = os.urandom(32)

102 if not self.cur.fetchone():
103     key = hashlib.pbkdf2_hmac('sha256', password.encode('utf-8'), salt, 100000)
104     sql_cmd = """
105         INSERT INTO Users
106         VALUES({id}, '{username}', '{password}', '{email}', '{admin}');
107     """
108
109     self.id += 1
110     sql_cmd = sql_cmd.format(id=self.id, username=username, password=key.hex(), email=email, admin=admin)
111
112     self.execute(sql_cmd)
113     self.commit()
114     return True
115 else:
116     return False
```

#### Notes:

- Line 5: Generates a string of 32 random bytes

### 1.2

Table: Users						
id	username	password	email	admin	publicKey	
Filter	Filter	Filter	Filter	Filter	Filter	
1	NULL	admin	1adf2a90d65b529dc01e01a6bf69d44a84ed4c1e8ba656...	admin@admin	1	None
2	NULL	owen	1630bc60266d19a6ad8b4e624e41670838d94368a36b...	admin@admin	0	{"alg": "RSA-...
3	NULL	cooper	65ae03947ad76ce00ca0605b20f7c3f2328dd7bd9fb987...	admin@admin	0	{"alg": "RSA-...

## 2.1

```
42 def run_server():
43     '''
44         run_server
45         Runs a bottle server
46     '''
47     #run(host=host, port=port, debug=debug)
48     run(server='gunicorn', host=host, port=port, debug=debug,
49         certfile='./certs/server.pem', keyfile='./certs/server.key')
50
```

Notes:

- Line 49: We stored the certificate and private key for the server in a folder called 'certs' for easy retrieval

## 2.2

```
[scoops@Coopers-Air template % python3 run.py]
Bottle v0.12.19 server starting up (using GunicornServer(certfile='./certs/server.pem', keyfile='./certs/server.key'))...
Listening on http://localhost:9033/
Hit Ctrl-C to quit.
```

Notes:

```
[2022-04-20 21:32:25 +1000] [72828] [INFO] Starting gunicorn 20.1.0
[2022-04-20 21:32:25 +1000] [72828] [INFO] Listening at: https://127.0.0.1:9033 (72828)
[2022-04-20 21:32:25 +1000] [72828] [INFO] Using worker: sync
[2022-04-20 21:32:25 +1000] [72830] [INFO] Booting worker with pid: 72830
```

3.1

← → ↺ 🏠

🔒 https://127.0.0.1:9033

📁 Apps 🌟 Bookmarks

🏠 Home 🔑 Login

🔒 Security

127.0.0.1:9033

🔒 Connection is secure


Your information (for example, passwords or credit card numbers) is private when it is sent to this site. [Learn more](#)

🔒 Certificate is valid

📧 WH Email ⚙️ Stafflink 👤 Sydney Student 🖨️ Canvas 📄 Ed 📖 My Library | Zotero 📜 Certificate 🔗 .crt to .pem 🖨️ Codepen 📺 Bein

Simple Student Templating Solutions

*Because the usability is more important than the back end for now.*



## 5.1

The screenshot shows the Chrome DevTools Application panel. The left sidebar lists various storage areas: Application (Manifest, Service Workers, Storage), Storage (Local Storage, Session Storage, IndexedDB, MyDatabase - https://127.0.0.1:9033, MyObjectStore, Web SQL, Cookies, Trust Tokens, Interest Groups), Cache (Cache Storage, Back/forward cache), and Background Services (Background Fetch). The main pane displays a table of IndexedDB data for 'MyObjectStore'. The table has two columns: '#', 'Key (Key path: "id")', and 'Value'. There is one entry with index 0, key '1', and a value object. The value object is expanded, showing 'id: 1' and 'key: CryptoKey {type: 'private', extractal...'. The bottom status bar indicates 'Total entries: 1'.

#	Key (Key path: "id")	Value
0	1	<pre>{id: 1, key: CryptoKey}   id: 1   key: CryptoKey {type: 'private', extractal</pre>

Total entries: 1

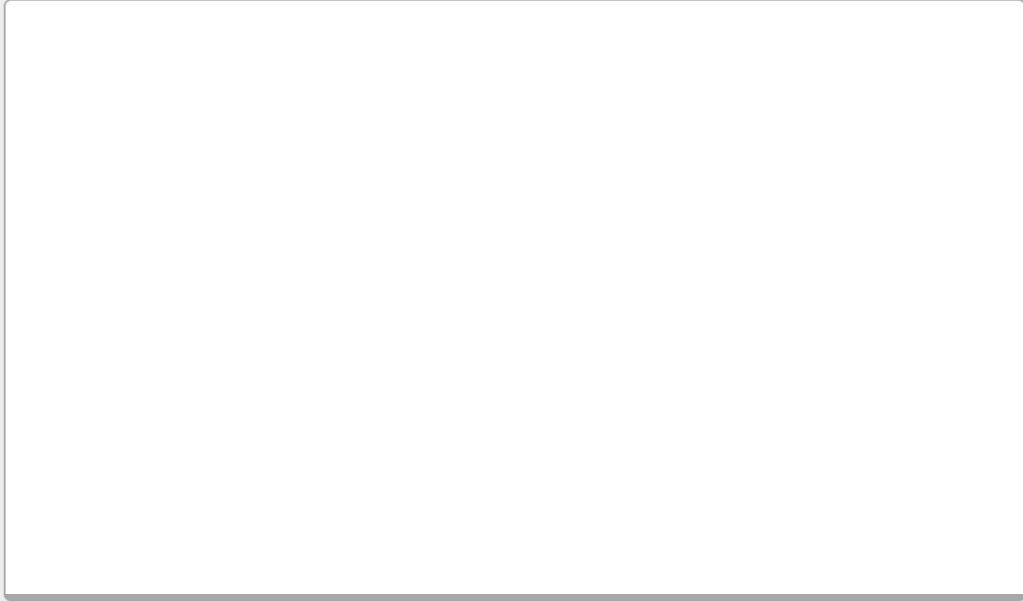
## 5.2

The screenshot shows a web application interface. At the top, it says 'Welcome, cooper'. Below this is a large, empty rectangular box. At the bottom, there is a form with the label 'To:' followed by a text input field containing the text 'owen'. To the right of the input field is an 'Enter' button.

Welcome, cooper

To:

Welcome, cooper

A large, empty rectangular box with a thin gray border, intended for the main body of a message or chat content.

Message: Hello Owen, how are you?

Send



Welcome, owen

From cooper: Hello Owen, how are you?

To:

Enter