

COMP3334 Project

End-to-end encrypted chat web application

Semester 2, 2023/2024

Overview

Nowadays, web services are the most common form of applications that users are exposed to. Web browsers become the most popular application on a computer that enables users to access those web services. Ensuring the security of web services is essential for the Internet. Moreover, privacy of communications is an important feature of modern times. Your job is to implement an end-to-end encrypted chat web application and secure various aspects of the website.



Objectives

1. Adapt a basic chat web application to become a secure E2EE chat web app
2. Comply with some of the requirements in [NIST Special Publication 800-63B](#) “Digital Identity Guidelines – Authentication and Lifecycle Management” for US federal agencies (which is also a reference for other types of systems)
3. Implement a secure MFA mechanism based on passwords and OTP (or FIDO2)
4. Encrypt communications between two users so that the server does not know the content of the messages (E2E encryption)
5. Protect communications in transit by configuring a modern TLS deployment
6. Package a docker image of your web app

Requirements (authentication)

1. From NIST Special Publication 800-63B:

1. Comply with all *SHALL* and *SHOULD* requirements from sections listed below

2. Use the following authenticators:

- User-chosen Memorized Secret (i.e., password/passphrase)
- and Single-Factor OTP Device (e.g., Google Authenticator)
 - or Single-Factor Cryptographic Device (e.g., Yubikey) if you have one
- and Look-Up Secrets (recovery keys)
- Comply with related requirements in §5.1 and §4.2.2
 - §5.1.1.2: “Memorized secrets SHALL be salted and hashed using a suitable one-way key derivation function”
 - See our Password Security lecture for an appropriate function
 - Memorized Secret Verifiers (§5.1.1.2)
 - Choose “**Passwords obtained from previous breach corpuses**” and refer to <https://haveibeenpwned.com/API/v3#PwnedPasswords> for the corpus to check against
 - §5.2.8 and §5.2.9 are automatically complied

Requirements (authentication)

1. From NIST Special Publication 800-63B:
 3. §5.2.2: Implement rate-limiting mechanisms AND image-based CAPTCHAs
 4. Implement new account registration and bind authenticators (OTP/Yubikey and recovery keys) at the same time
 - Optional: provide a way to change authenticators after account registration
 5. §7.1: Implement proper session binding requirements
6. Exceptions:
 - ~~OTP authenticators — particularly software-based OTP generators — SHOULD discourage and SHALL NOT facilitate the cloning of the secret key onto multiple devices.~~
 - Google Authenticator and related apps are OK

Requirements (E2EE chat)

2. Once users are logged in, secure chat messages between two users in a way so that the server cannot decrypt the messages
 1. Use the **ECDH key exchange** protocol to establish a shared secret between two users
 - Leverage the [WebCrypto API](https://webkit.org/demos/webcrypto/ecdh.html), see demo <https://webkit.org/demos/webcrypto/ecdh.html>
 - Exchanged information during the key exchange can be sent through the server
 - The server is trusted not to modify messages of the key exchange
 - Choose **P-384** as the underlying curve
 2. **Derive two 256-bit AES-GCM encryption keys** and **two 256-bit MAC keys** from the shared secret using **HKDF-SHA256**
 - One key for encryption between user1 to user2, and another one from user2 to user1
 - Using WebCrypto API again, see <https://developer.mozilla.org/en-US/docs/Web/API/HkdfParams>
 - The **salt** should be unique so another key derivation in the future produces different keys, use for instance a counter starting at 1
 - The **info** parameter should represent the current context (e.g., “CHAT_KEY_USER1to2” for the key for user1→user2, and “CHAT_MAC_USER1to2” for the MAC key for user1→user2)

Requirements (E2EE chat)

2. Once users are logged in, secure chat messages between two users in a way so that the server cannot decrypt the messages
3. Messages will be encrypted using **AES in GCM mode**
 - 96-bit IVs are **counters** representing the number of messages encrypted with the same key
 - Note: GCM does not require unpredictable IVs, but unique IVs
 - Send the IV together with the ciphertext to the recipient
 - As a recipient, verify that $IV_i > IV_{i-1}$ to prevent replay attacks
 - Protect the IV with **HMAC-SHA256** using the derived MAC key to prevent the attacker from choosing IVs
 - **Associated data** should reflect the current context (e.g., "CHAT_MSG_USER1to2")
 - Authentication tags should be 128 bits
4. Store all key material in the HTML5 Local Storage of the browser to be retrieved after the browser is reopened
5. Display the history of previous messages being exchanged + new messages
 - If Local Storage has been cleared, previous messages cannot be decrypted, show warning

Requirements (E2EE chat)

2. Once users are logged in, secure chat messages between two users in a way so that the server cannot decrypt the messages
6. All symmetric keys and IVs should be **re-derived** from the shared secret when **user clicks on a “Refresh” button** in the chat (not the browser refresh button), using a new salt
 - The participant that requests a change should inform the other party with a special message composed of the last IV that has been used, the string “change”, altogether protected with the **old MAC key** AND the **new MAC key**
 - Two different MACs over the message
 - The other party should verify the old MAC before processing the message, then derive new keys and verify again the new MAC before accepting the new keys
 - Both parties should show a message “**Keys changed**” in the chat history
 - **Old keys should be kept** to decrypt older messages when the browser is reopened, you should identify which set of keys to use for a given message based on the preceding values sent during the key exchange (i.e., keep track of user public keys)
 - Key exchange messages older than a minute should not be considered as a fresh key exchange to engaged into

Requirements (E2EE chat)

2. Once users are logged in, secure chat messages between two users in a way so that the server cannot decrypt the messages
7. When the Local Storage is cleared, or when there is no shared secret for a given recipient, the sender should initiate the ECDH key exchange using a special message and the recipient should engage in the key exchange even when there had been a shared secret previously established
8. Chat messages should be encoded using **UTF-8**, and network messages between users should be formatted in **JSON** using your own schema (e.g., {"type":"ECDH", "key":"..."}, {"type":"msg", "ciphertext":"...", "IV":"...", "MAC":"..."})
9. Use console.log() to **log all crypto operations** (including key, IV, plaintext, etc.)
 - It should be visually obvious that IVs are not reused, keys change when needed (see next requirements), etc.
10. The chat app should be protected against cross-site request forgery (**CSRF**), cross-site scripting (**XSS**), and **SQL** injection attacks

Requirements (TLS)

3. Communications should be encrypted in transit using TLS with the following configuration:

- Reuse Mozilla's "modern" configuration for nginx, and change it as needed:

- <https://ssl-config.mozilla.org/>

1. TLS version 1.3 only
2. x25519 Elliptic Curve Group only
3. TLS_CHACHA20_POLY1305_SHA256 cipher suite only
4. No OCSP stapling (since you will use a self-signed CA certificate)
5. HSTS for one week
6. TLS certificate requirements:
 1. X.509 version 3
 2. ECDSA public key over P-384
 3. SHA384 as hashing algorithm for signature
 4. CA flag (critical): false
 5. Key Usage (critical) = Digital Signature
 6. Extended Key Usage = Server Authentication
 7. Include both Subject Key Identifier and Authority Key Identifier
 8. Validity period = 90 days

Requirements (TLS)

3. Communications should be encrypted in transit using TLS with the following configuration:

7. The website should be hosted at

[https://group-\[your-group-number\].comp3334.xavier2dc.fr:8443/](https://group-[your-group-number].comp3334.xavier2dc.fr:8443/)

- Group #10 will be at group-10.comp3334.xavier2dc.fr

8. All subdomains *.comp3334.xavier2dc.fr will redirect to 127.0.0.1

- You can effectively use “group-X.comp3334.xavier2dc.fr” instead of “localhost”

- If you do not host the docker container on localhost, add a manual entry in your hosts file

- Linux: /etc/hosts
- Windows: C:\Windows\System32\drivers\etc\hosts

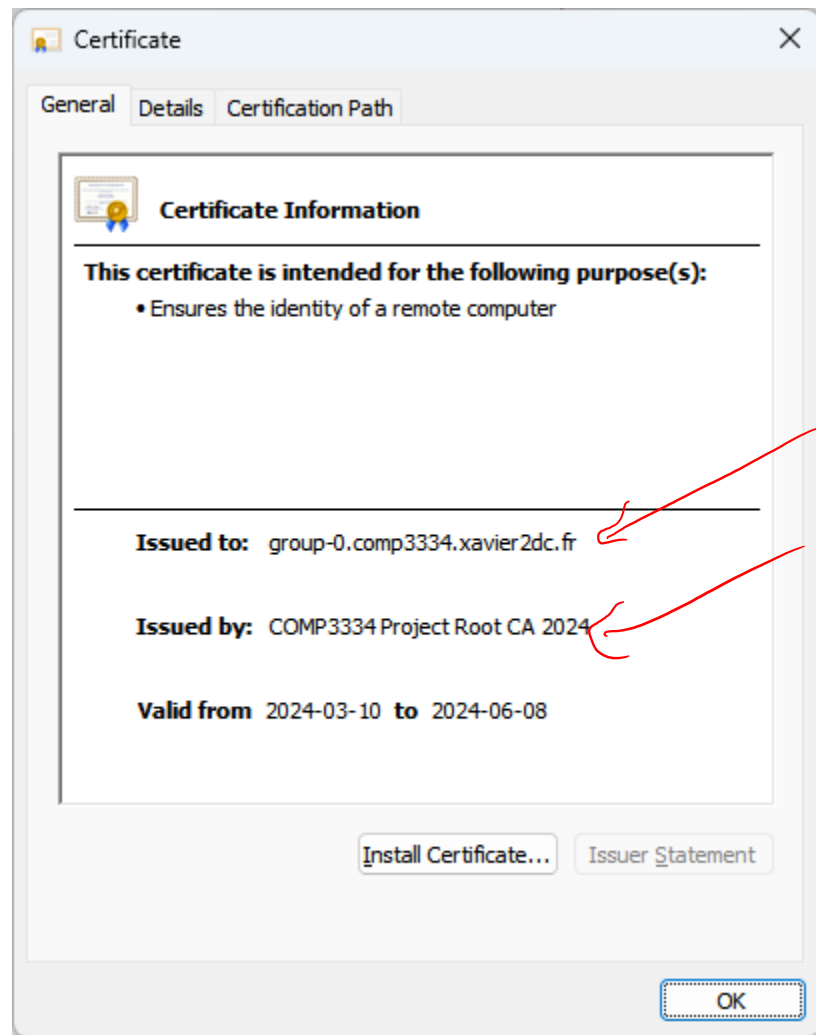
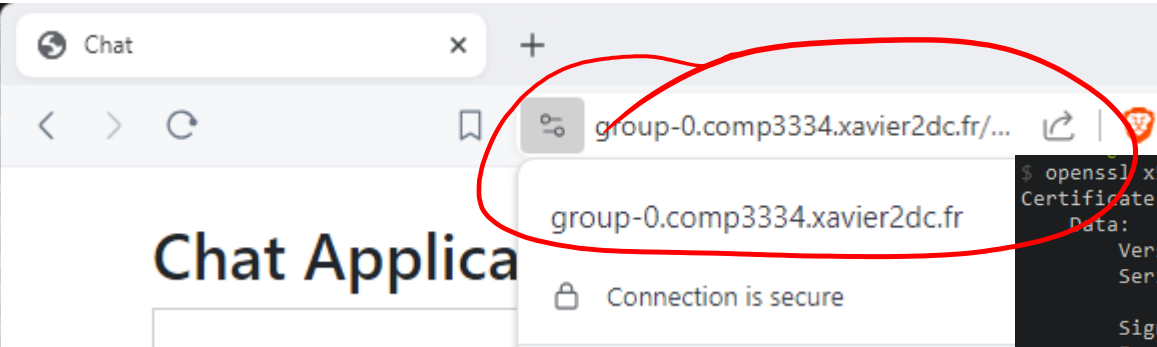
```
> group-1234.comp3334.xavier2dc.fr
Server:  dns.google
Address:  8.8.8.8

Non-authoritative answer:
Name:     group-1234.comp3334.xavier2dc.fr
Address:  127.0.0.1
```

9. **Issue the certificate from the given CA certificate and private key**



- Use the domain name corresponding to your group
- Domain should appear as both **Common Name** and **Subject Alternative Name**

10. The CA certificate is domain-constrained to subdomains of comp3334.xavier2dc.fr, meaning you can safely trust it on your computer (nobody can generate valid certificates for other domains)



```
$ openssl x509 -in web.crt -noout -text
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      5a:c3:c6:4c:5d:ba:a4:67 8a:ac:c4:a6:12:a0:82:99:43:e8:69:7c
    Signature Algorithm: ecdsa-with-SHA384
    Issuer: C=HK, O=The Hong Kong Polytechnic University, OU=Department of Computing, CN=COMP3334 Project Root CA 2024
    Validity
      Not Before: Mar 10 07:06:29 2024 GMT
      Not After : Jun  8 07:06:29 2024 GMT
    Subject: C=HK, CN=group-0.comp3334.xavier2dc.fr
    Subject Public Key Info:
      Public Key Algorithm: id-ecPublicKey
      Public-Key: (384 bit)
      pub:
        04:48:69:1d:df:23:b9:b4:3a:f0:24:2b:72:73:fa:
        e7:2a:e0:44:8c:c0:8d:70:6f:71:4e:07:4c:2e:17:
        82:2b:c5:45:36:0b:84:04:85:b3:67:2a:6e:39:c8:
        73:7b:4d:68:86:45:a0:53:cb:0e:3c:15:3b:65:57:
        54:c3:b4:86:1d:98:61:d2:f7:94:63:06:db:a6:78:
        2f:19:99:49:17:2e:85:35:45:95:ad:ba:bf:ef:ef:
        57:00:0b:00:be:9e:9b
      ASN1 OID: secp384r1
      NIST CURVE: P-384
    X509v3 extensions:
      X509v3 Subject Key Identifier:
        A0:B0:31:7A:00:3A:AD:0A:E6:8A:EF:96:42:1A:E8:81:C5:A1:62:DC
      X509v3 Authority Key Identifier:
        3B:4E:1B:40:FD:B5:1C:FF:7C:33:DB:B6:FB:AF:3C:BC:EC:24:2B:CE
      X509v3 Basic Constraints: critical
        CA:FALSE
      X509v3 Key Usage: critical
        Digital Signature
      X509v3 Extended Key Usage:
        TLS Web Server Authentication
      X509v3 Subject Alternative Name:
        DNS:group-0.comp3334.xavier2dc.fr
    Signature Algorithm: ecdsa-with-SHA384
    Signature Value:
      30:81:87:02:42:01:60:3f:95:7e:bd:9f:74:86:5e:4a:df:0d:
      29:b5:93:63:6c:62:bf:f3:b4:fe:88:59:de:8a:bb:db:e5:28:
      86:39:57:9f:ac:f8:07:0e:76:20:a3:db:29:bc:e9:96:db:bf:
      ed:b5:72:77:e2:6f:0c:aa:8d:19:8e:4c:a5:87:18:54:0b:02:
      41:54:6e:bc:ed:04:da:09:14:0a:63:e2:db:ec:c1:ca:d0:0d:
      ea:c2:1c:65:99:d9:11:d4:f5:ad:2c:80:df:99:43:a2:e5:0e:
      be:ca:0a:1e:f6:6b:7d:03:7e:4c:95:ee:c8:a4:f4:d8:0c:f5:
      96:60:9c:64:65:c8:50:e8:7d:33:70:45
```

Simple Chat Demo

1. Deploy the docker container using the following line within the folder that contains the docker-compose.yaml file:
`$ sudo docker-compose up -d`
2. So far, the chat app works over plain HTTP on port 8080, access it at:
<http://group-0.comp3334.xavier2dc.fr:8080>
3. Open a new private window of your browser and access the website again
 1. Chrome:  New private window Ctrl+Shift+N
 2. Firefox:  New private window Ctrl+Shift+P
4. Login as Alice (password: password123) on the first window
5. Login as Bob (password: password456) on the second (private) window
6. Select Bob as contact from Alice's chat, select Alice as contact from Bob's chat
7. Send messages each other!
8. When modifying the server-side (app.py) or client-side (login.html, chat.html), simply restart the docker container, you do not need to rebuild the container:
`$ sudo docker restart [you-container-name]-webapp-1`

Areas of assessments

1. Explanations of your solution and design [50%]
 - Provide list of features/requirements implemented
 - Describe how your solution works, especially explain how user passwords are stored, verified, which libraries do you use, how key materials are derived, how do you store them, their size, how do you generate the domain certificate, etc.
 - Show autonomy and creativity when requirements allow
2. Implementation of your solution & demo [50%]
 - Follow proper coding style, write informative comments, give concise and relevant variable names, respect indentation, stay consistent in style
 - Make things work!

Submission

- Submit a ZIP'd file containing:
 1. Modified chat app docker-compose stack
 - “sudo docker-compose up -d” should work!
 - Accessing <https://group-X.comp3334.xavier2dc.fr:8443/> should work with a valid certificate issued by the given CA
 - Group number is the one you registered on Blackboard
 2. PDF report
 3. 8-minute video with a demonstration of your solution
 - User registration + new chat with existing user + refresh website & reload chat
 4. Statement of individual contributions
 - Who did what, how much % of the work does that represent?
 - Format will be given to you later
- **Deadline for submission is Sunday, April 14 @ 23:59 (hard deadline)**

Questions?

Technical questions:

- CUI Bowen bowen.cui@connect.polyu.hk

Administrative questions:

- LYU Xinqi xinqi.lyu@connect.polyu.hk

FAQ

1. Can I use a library?
 - Depends, does it replace the whole chat protocol with a better and secure chat? Then, no. You still need to implement a secure chat protocol.
 - Does the library implement part of the requirements (e.g., proper session management, OTP, hashing algorithm, etc.)? Then, yes.
2. How can I rebuild the docker container if I need to modify, say, the nginx config?
 1. `docker-compose down -v`
 2. `docker-compose build --no-cache`
 3. `docker-compose up -d`
3. How can I debug errors?
 - `docker logs [your-container]`

FAQ

4. How does the web chat application work?
 1. It is written in Python using Flask
 2. It is running behind the WSGI server Gunicorn
 3. Which is running behind the reverse proxy nginx (which should provide TLS)
 4. The front-end is written in HTML and Javascript
 5. The server app writes messages into a MySQL database