ISS CW2

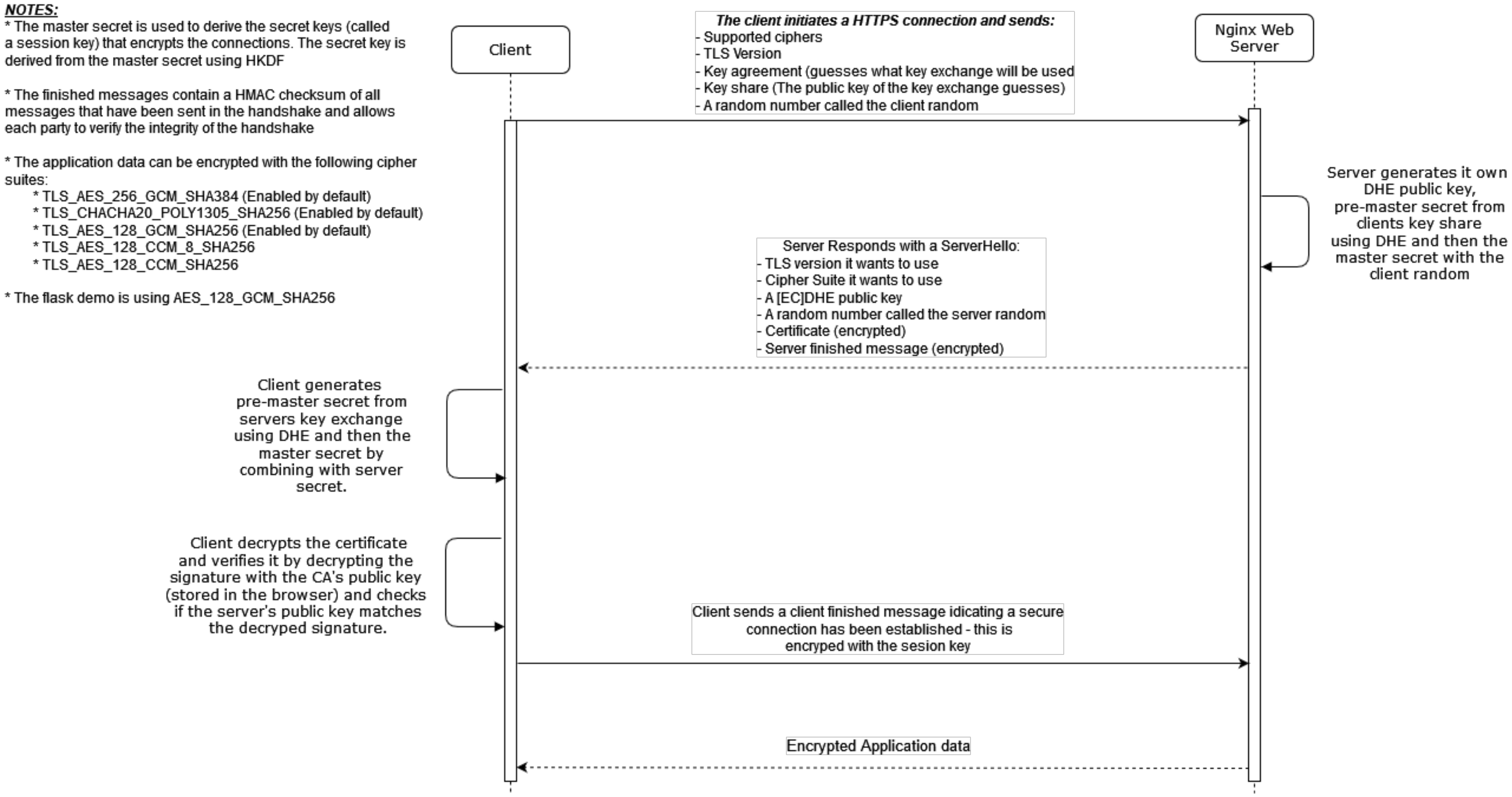
u2240264

Contents

1 Context & Assumptions

In the code file (iss_cw2/app/app.py) you will find a list of assumptions and context for the demo app.
The README file contains instructions to run the application locally
USER_ACCOUNTS contains the demo Google accounts you can use to access each role of the application running at https://iss.oscarsharpe.me

2 TLS Handshake diagram



NOTES:
* The master secret is used to derive the secret keys (called a session key) that encrypts the connections. The secret key is derived from the master secret using HKDF

* The finished messages contain a HMAC checksum of all messages that have been sent in the handshake and allows each party to verify the integrity of the handshake

* The application data can be encrypted with the following cipher suites:
    * TLS_AES_256_GCM_SHA384 (Enabled by default)
    * TLS_CHACHA20_POLY1305_SHA256 (Enabled by default)
    * TLS_AES_128_GCM_SHA256 (Enabled by default)
    * TLS_AES_128_CCM_8_SHA256
    * TLS_AES_128_CCM_SHA256

* The flask demo is using AES_128_GCM_SHA256

The client initiates a HTTPS connection and sends:
- Supported ciphers
- TLS Version
- Key agreement (guesses what key exchange will be used
- Key share (The public key of the key exchange guesses)
- A random number called the client random

Server generates it own DHE public key, pre-master secret from clients key share using DHE and then the master secret with the client random

Server Responds with a ServerHello:
- TLS version it wants to use
- Cipher Suite it wants to use
- A [EC]DHE public key
- A random number called the server random
- Certificate (encrypted)
- Server finished message (encrypted)

Client generates pre-master secret from servers key exchange using DHE and then the master secret by combining with server secret.

Client decrypts the certificate and verifies it by decrypting the signature with the CA's public key (stored in the browser) and checks if the server's public key matches the decryped signature.

Client sends a client finished message idicating a secure connection has been established - this is encryped with the sesion key

Encrypted Application data

Figure 1: Sequence Diagram for TLS1.3 used in the flask app (Rescorla, 2018; Cloudflare, 2023; Nohe, 2023; Thakkar, 2023)

3 Login Flow diagram

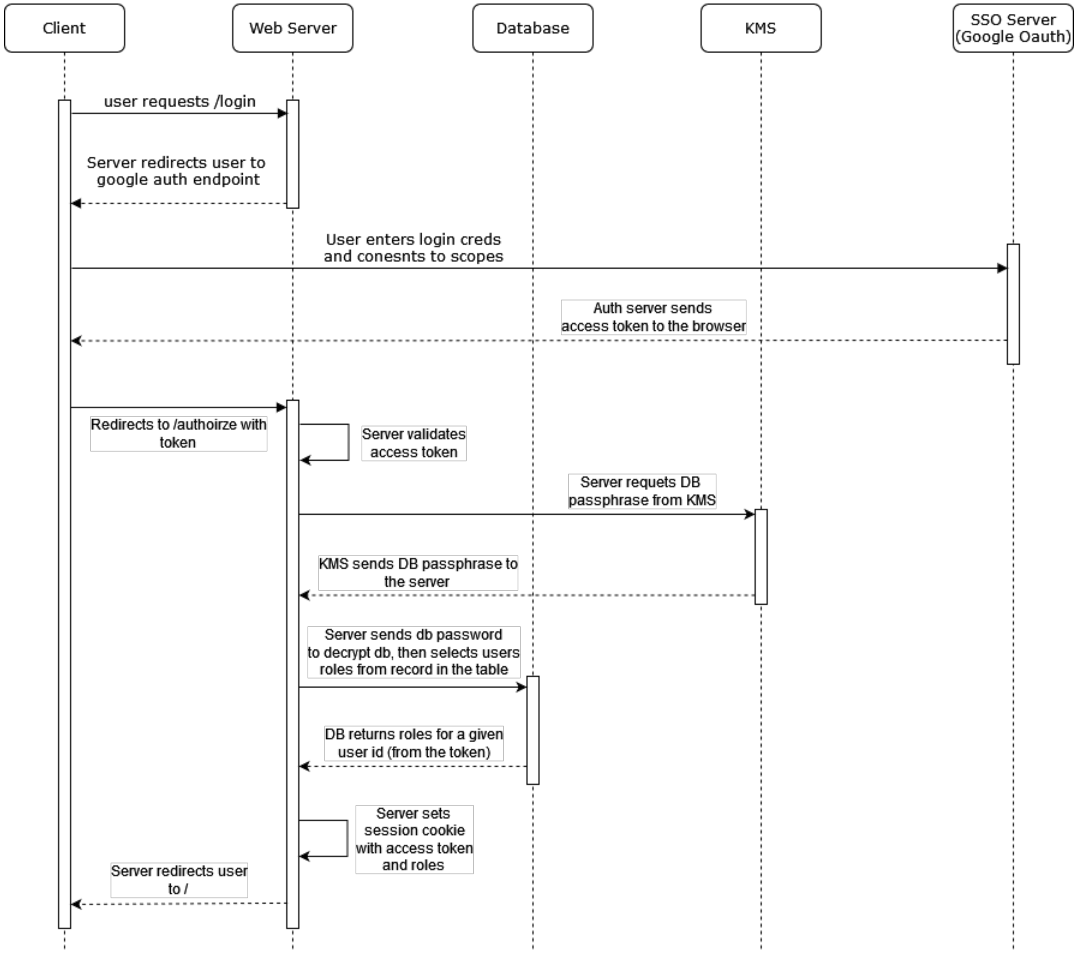
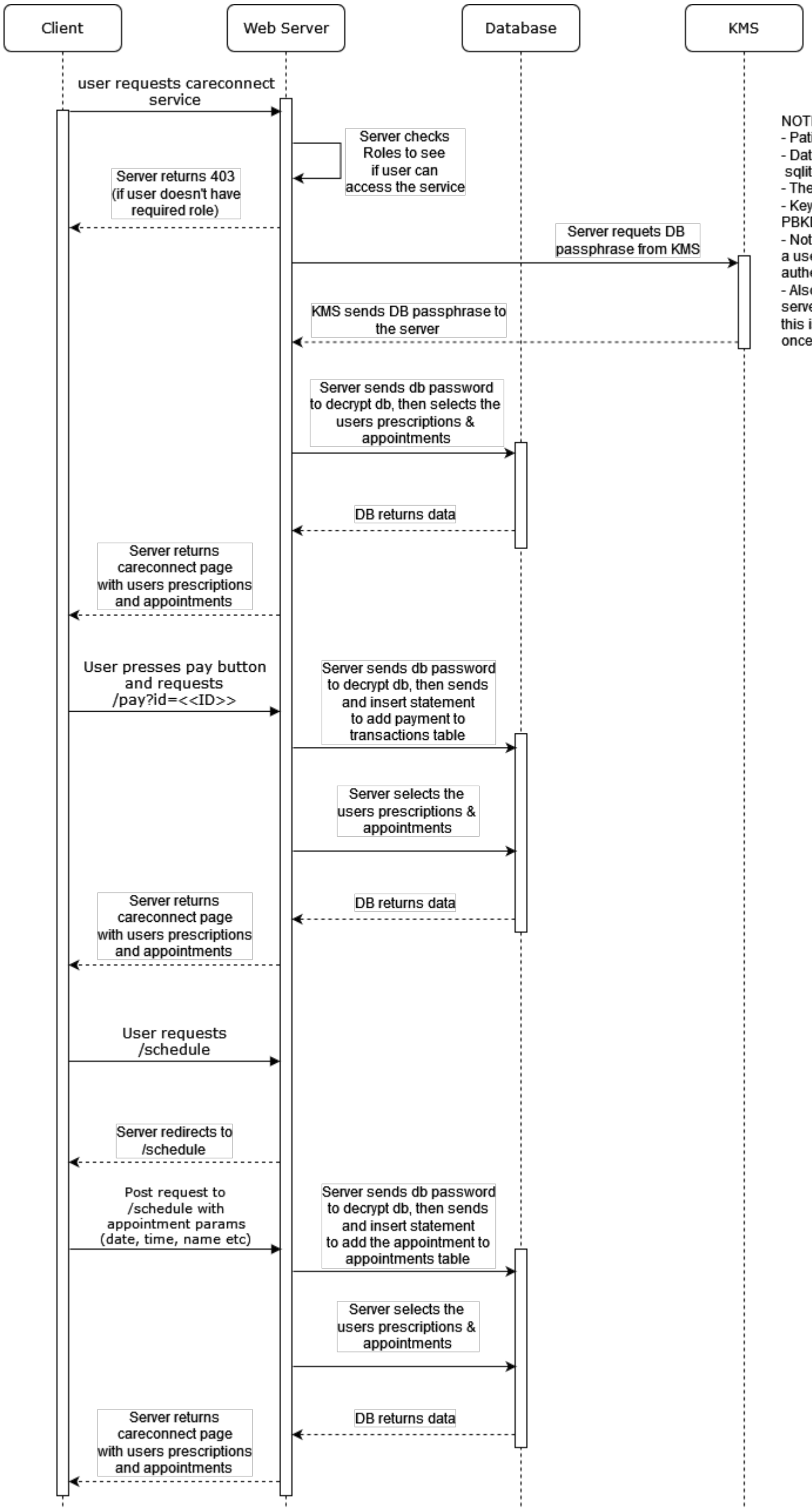

Client
Web Server
Database
KMS
SSO Server (Google Oauth)
user requests /login
Server redirects user to google auth endpoint
User enters login creds and conesnts to scopes
Auth server sends access token to the browser
Redirects to /authoirze with token
Server validates access token
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects users roles from record in the table
DB returns roles for a given user id (from the token)
Server sets session cookie with access token and roles
Server redirects user to /

Figure 2: Sequence Diagram for the login flow of the demo. All requests to the web & oauth server made over HTTPS, see Figure 1 (OneLogin, 2019) (Hardt, 2012, §1.2, §4.2)

4 CareConnect Diagram


Client
Web Server
Database
KMS
user requests careconnect service
Server checks Roles to see if user can access the service
Server returns 403 (if user doesn't have required role)
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects the users prescriptions & appointments
DB returns data
Server returns careconnect page with users prescriptions and appointments
User presses pay button and requests /pay?id=<<ID>>
Server sends db password to decrypt db, then sends and insert statement to add payment to transactions table
Server selects the users prescriptions & appointments
DB returns data
Server returns careconnect page with users prescriptions and appointments
User requests /schedule
Server redirects to /schedule
Post request to /schedule with appointment params (date, time, name etc)
Server sends db password to decrypt db, then sends and insert statement to add the appointment to appointments table
Server selects the users prescriptions & appointments
DB returns data
Server returns careconnect page with users prescriptions and appointments

NOTES:
- Patient role is required for this service
- Database uses sqlcipher, a fork of
 sqlite3
- The database is encrypted with AES-256 CBC
- Key is derived using the passphrase using
PBKDF2-HMAC-SHA512
- Not shown in the diagram is that every time
a user accesses a route, the server checks
authentication and user roles
- Also before each database accesses the
server retrieved the key from the KMS
this is omitted from this diagram to save space (only shown
once in the diagram)

Figure 3: Sequence diagram for the careconnect system. A user needs to be logged in with the patient role to use to service, see Figure 2

5 FinCare Diagram

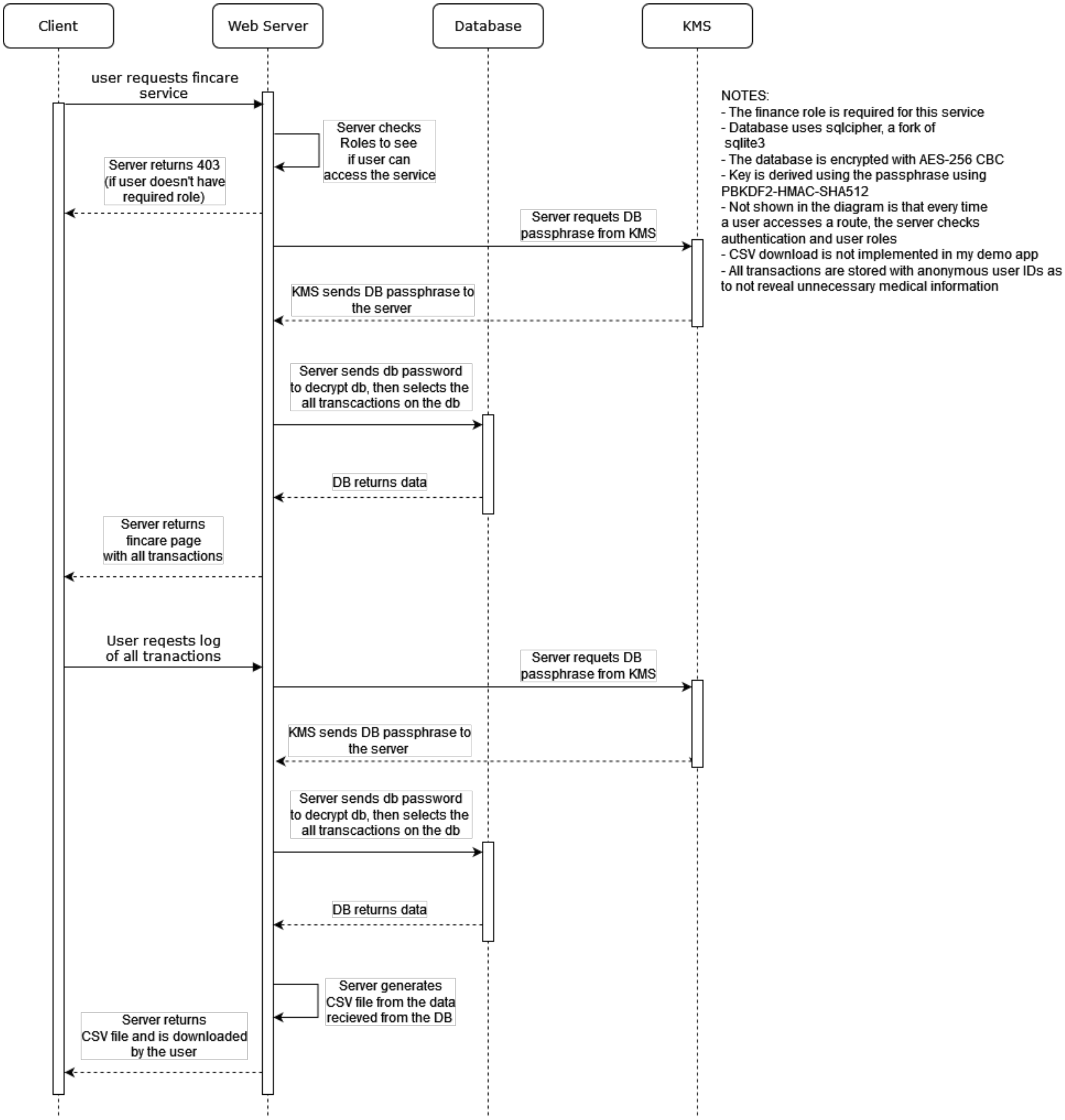
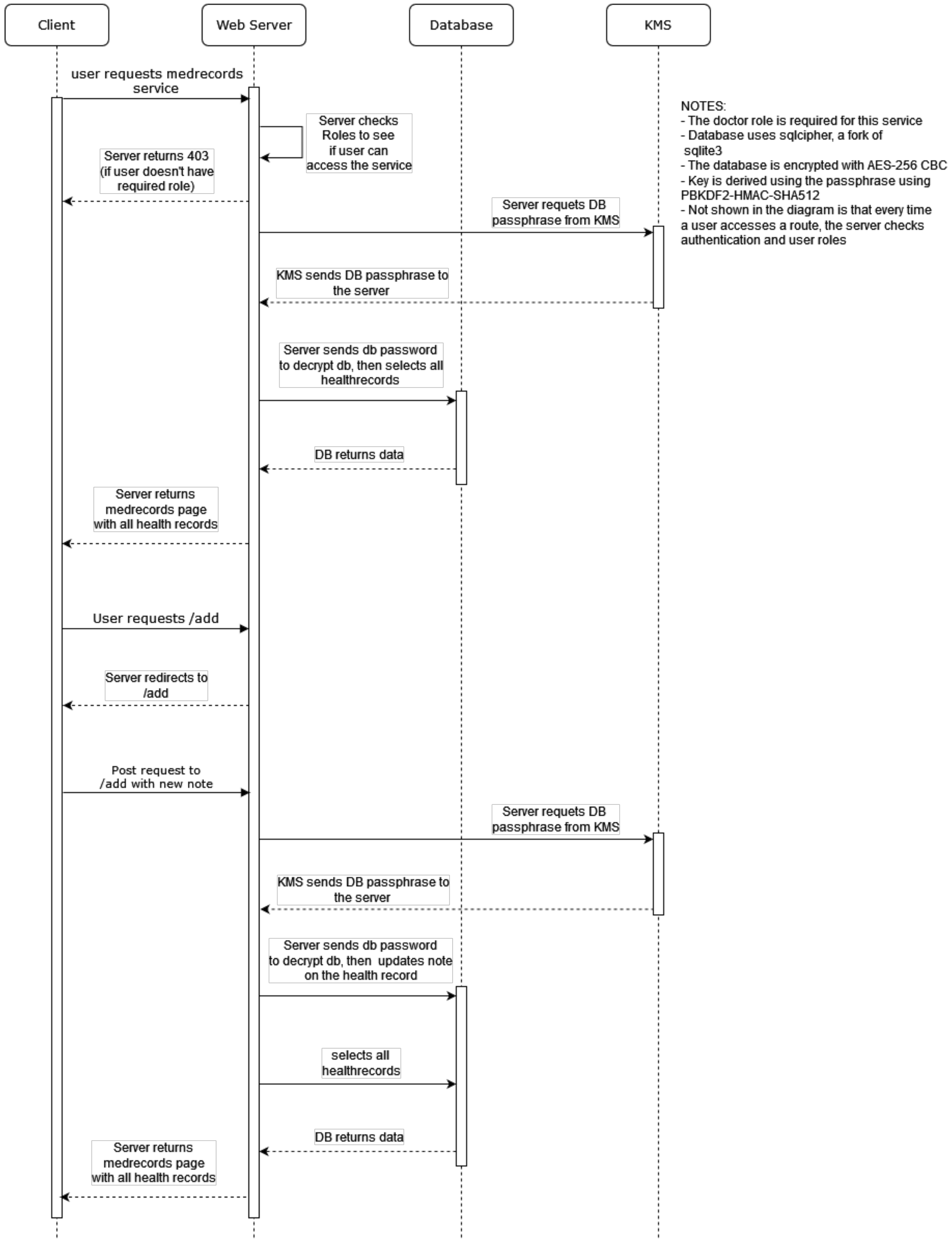

Client
Web Server
Database
KMS
user requests fincare service
Server checks Roles to see if user can access the service
Server returns 403 (if user doesn't have required role)
NOTES:
- The finance role is required for this service
- Database uses sqlcipher, a fork of sqlite3
- The database is encrypted with AES-256 CBC
- Key is derived using the passphrase using PBKDF2-HMAC-SHA512
- Not shown in the diagram is that every time a user accesses a route, the server checks authentication and user roles
- CSV download is not implemented in my demo app
- All transactions are stored with anonymous user IDs as to not reveal unnecessary medical information
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects the all transcactions on the db
DB returns data
Server returns fincare page with all transactions
User reqests log of all tranactions
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects the all transcactions on the db
DB returns data
Server generates CSV file from the data recieved from the DB
Server returns CSV file and is downloaded by the user

Figure 4: Sequence diagram for the FinCare system. A user needs to be logged in with the finance role to use to service, see Figure 2

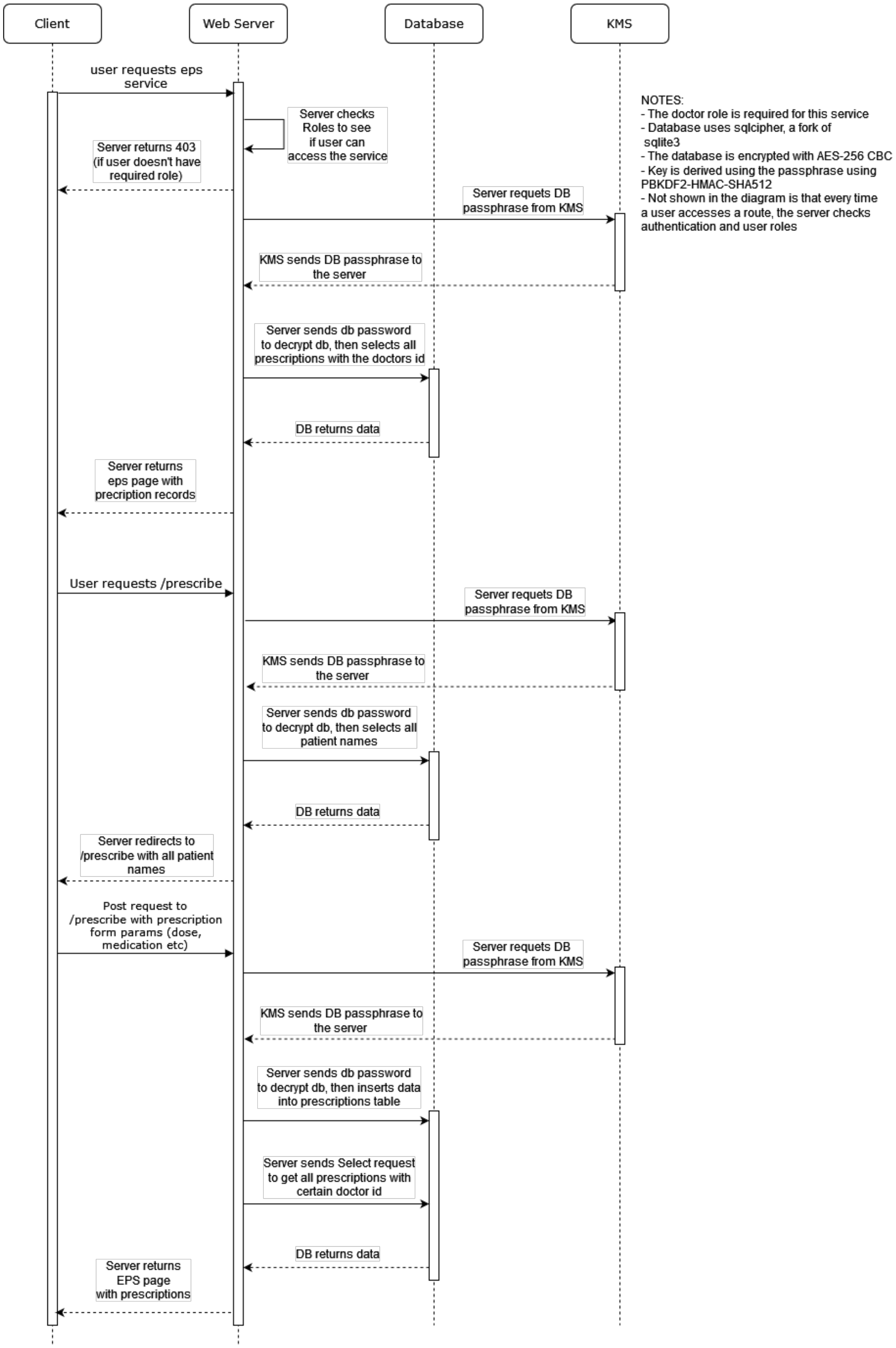
6 MedRecords Diagram


Client
Web Server
Database
KMS
user requests medrecords service
Server checks Roles to see if user can access the service
Server returns 403 (if user doesn't have required role)
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects all healthrecords
DB returns data
Server returns medrecords page with all health records
User requests /add
Server redirects to /add
Post request to /add with new note
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then updates note on the health record
selects all healthrecords
DB returns data
Server returns medrecords page with all health records

NOTES:
- The doctor role is required for this service
- Database uses sqlcipher, a fork of sqlite3
- The database is encrypted with AES-256 CBC
- Key is derived using the passphrase using PBKDF2-HMAC-SHA512
- Not shown in the diagram is that every time a user accesses a route, the server checks authentication and user roles

Figure 5: Sequence diagram for the MedRecords system. A user needs to be logged in with the doctor role to use to service, see Figure 2

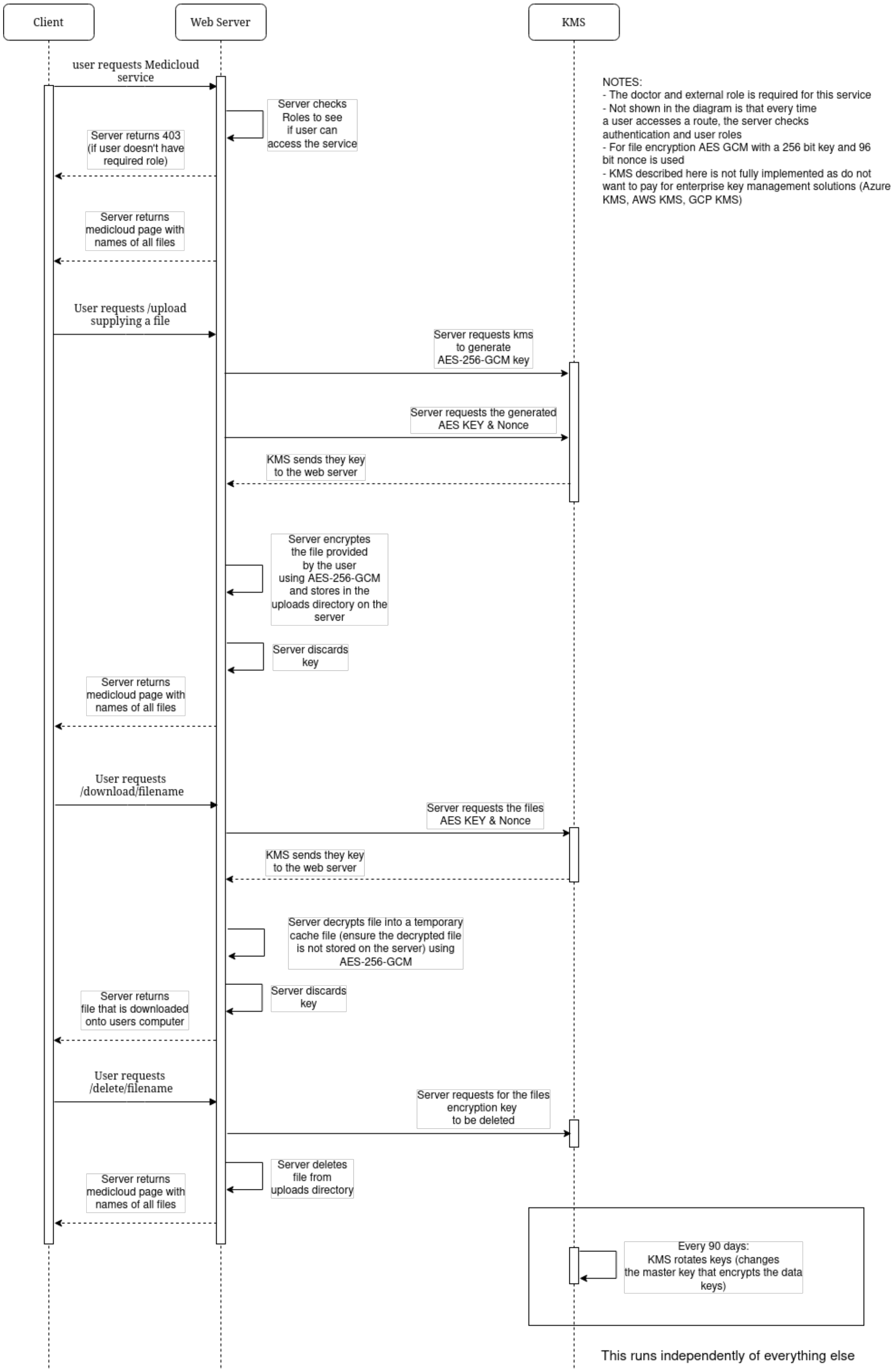
7 EPS Diagram


Client
Web Server
Database
KMS
user requests eps service
Server checks Roles to see if user can access the service
Server returns 403 (if user doesn't have required role)
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects all prescriptions with the doctors id
DB returns data
Server returns eps page with precription records
User requests /prescribe
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then selects all patient names
DB returns data
Server redirects to /prescribe with all patient names
Post request to /prescribe with prescription form params (dose, medication etc)
Server requets DB passphrase from KMS
KMS sends DB passphrase to the server
Server sends db password to decrypt db, then inserts data into prescriptions table
Server sends Select request to get all prescriptions with certain doctor id
DB returns data
Server returns EPS page with prescriptions

NOTES:
- The doctor role is required for this service
- Database uses sqlcipher, a fork of sqlite3
- The database is encrypted with AES-256 CBC
- Key is derived using the passphrase using PBKDF2-HMAC-SHA512
- Not shown in the diagram is that every time a user accesses a route, the server checks authentication and user roles

Figure 6: Sequence diagram for the EPS system. A user needs to be logged in with the doctor role to use to service, see Figure 2

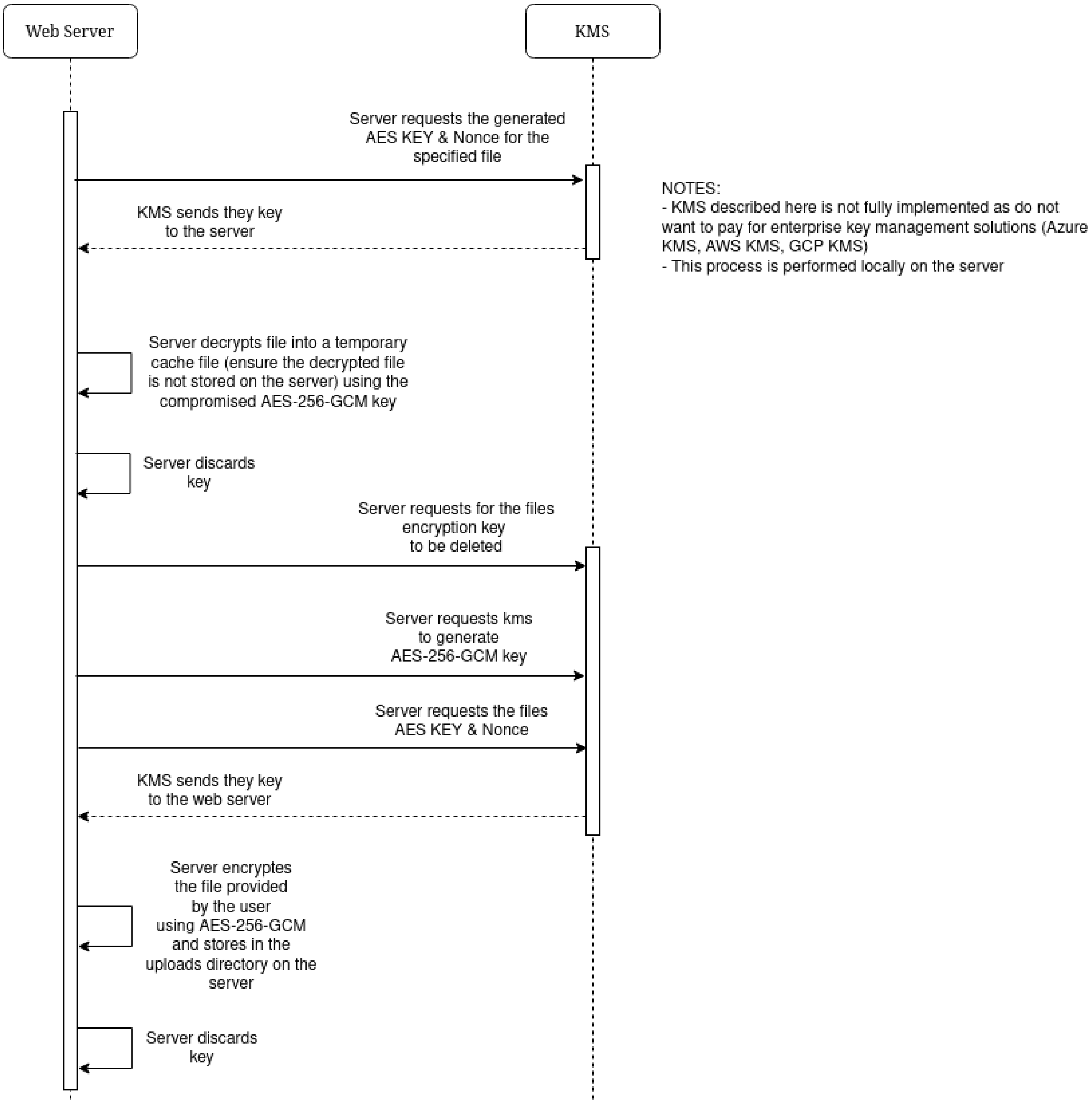
8 Medicloud Diagram


Client
Web Server
KMS
user requests Medicloud service
Server checks Roles to see if user can access the service
Server returns 403 (if user doesn't have required role)
Server returns medicloud page with names of all files
User requests /upload supplying a file
Server requests kms to generate AES-256-GCM key
Server requests the generated AES KEY & Nonce
KMS sends they key to the web server
Server encryptes the file provided by the user using AES-256-GCM and stores in the uploads directory on the server
Server discards key
Server returns medicloud page with names of all files
User requests /download/filename
Server requests the files AES KEY & Nonce
KMS sends they key to the web server
Server decrypts file into a temporary cache file (ensure the decrypted file is not stored on the server) using AES-256-GCM
Server discards key
Server returns file that is downloaded onto users computer
User requests /delete/filename
Server requests for the files encryption key to be deleted
Server deletes file from uploads directory
Server returns medicloud page with names of all files
Every 90 days: KMS rotates keys (changes the master key that encrypts the data keys)
This runs independently of everything else

NOTES:
- The doctor and external role is required for this service
- Not shown in the diagram is that every time a user accesses a route, the server checks authentication and user roles
- For file encryption AES GCM with a 256 bit key and 96 bit nonce is used
- KMS described here is not fully implemented as do not want to pay for enterprise key management solutions (Azure KMS, AWS KMS, GCP KMS)

Figure 7: Sequence diagram for the Medicloud system (AWS, 2024) A user needs to be logged in with the doctor or external role to use to service, see Figure 2

9 Key rotation if a key is compromised


Web Server
KMS
Server requests the generated AES KEY & Nonce for the specified file
KMS sends they key to the server
Server decrypts file into a temporary cache file (ensure the decrypted file is not stored on the server) using the compromised AES-256-GCM key
Server discards key
Server requests for the files encryption key to be deleted
Server requests kms to generate AES-256-GCM key
Server requests the files AES KEY & Nonce
KMS sends they key to the web server
Server encryptes the file provided by the user using AES-256-GCM and stores in the uploads directory on the server
Server discards key

NOTES:
- KMS described here is not fully implemented as do not want to pay for enterprise key management solutions (Azure KMS, AWS KMS, GCP KMS)
- This process is performed locally on the server

Figure 8: Sequence diagram for the *data_key_rotate.py* script

References

AWS (Jan. 2024). *Rotating AWS KMS Keys - AWS Key Management Service*. URL: https://docs.aws.amazon.com/kms/latest/developerguide/rotate-keys.html.

Cloudflare (Oct. 2023). URL: https://www.cloudflare.com/learning/ssl/what-happens-in-a-tls-handshake/.

Hardt, Dick (Oct. 2012). *The OAuth 2.0 Authorization Framework*. RFC 6749. DOI: 10.17487/RFC6749. URL: https://www.rfc-editor.org/info/rfc6749.

Nohe, Patrick (Feb. 2023). *Taking a closer look at the SSL handshake*. URL: https://www.thesslstore.com/blog/explaining-ssl-handshake/.

OneLogin (Jan. 2019). *Developer overview of OpenID connect*. URL: https://developers.onelogin.com/openid-connect.

Rescorla, Eric (Aug. 2018). *The Transport Layer Security (TLS) Protocol Version 1.3*. 8446. 160 pp. DOI: 10.17487/RFC8446. URL: https://www.rfc-editor.org/info/rfc8446.

Thakkar, Jay (Mar. 2023). *TLS 1.3 handshake: Taking a closer look*. URL: https://www.thesslstore.com/blog/tls-1-3-handshake-tls-1-2/.