Programming Final Project: Hospital App

Done By:

Hajar El Boutahiri

Pham Vu Van Thanh

Supervised By:

Professor Marko Helenius

COMP.SEC.300: Secure Programming

Academic Year: 2024 - 2025

Table of Content

I. General Description

Our project developed a web application for a hospital to manage appointments between patients and doctors including booking and viewing personal appointments and modifying personal profiles.

The application uses Role Based Access Control which supports 5 roles: default admin (created with the database), admin, doctor, patient, and user. Each role will have a different view after logging in based on their privileges. By default, a new sign-up user's role will be set as "user" and does not have the privilege to book an appointment. The admin needs to assign them a role as "patient", "doctor" or "admin" to be allowed to interact with code functions and endpoints.

Notice: default admin account can be logged in with username: **admin**, password: **admin**

II. Structure Of The Program

This project has been implemented from scratch. We have worked on a full stack project.

For Our Frontend:

- Angular 19.2.6
  - Languages & Technologies:
    - HTML
    - CSS
    - TypeScript
  - Structure
    - Components => Different Components to Build The User Interface
    - Services => Define Reusable Logic and Data Access
    - Models => Define Data Structure and types
    - Guard => Control Access Based on Token Validity
    - Interceptor => Add and Remove Authentication Token From HTTP Requests
    - Server.ts => Set Up Express Server For Server-Side rendering
    - App.routes.ts => Client-Side Routing Configuration

For Our Backend

- FastAPI 0.115.12
  - Language:
    - Python
  - Structure:
    - Core => Shared Utilities & Logic
    - Models => Define Database Tables (Entities)
    - Schemas => Define Pydantic Models
    - APIs => Handle Endpoints Routing
    - Main.py => FastAPI initialization
    - Database.py => Database Set up

For Our Database:

- SQLITE 3.45.3
    - SQL (Structured Query Language)

For Our Security Testing:

- Docker
    - Docker Compose

III. Secure Programming Solutions

A. OWASP Top Ten 2025

1. A01:2021-Broken Access Control

1.1. Role Based Access Control (RBAC) Implementation

We have set 5 distinct roles. Allowed actions, page and data viewed depends on the specific role.

- Default Admin Role
    - Specifications
        - The default admin is created by default
        - The default admin can't be deleted
        - Only 1 default admin is present
    - Allowed Actions
        - Update only its own personal data
        - Assign a user to patient/ doctor/ normal admin role
        - Deactivate a patient/ doctor/ normal admin
        - Book an appointment on behalf of a patient or doctor
        - Update Appointment Information
            - Update the appointment status (to Cancelled, Confirmed, Scheduled, Completed, in progress) based on the pre-set rules
            - Update the appointment date, time and description
    - Viewed Data
        - All system data can be viewed
            - Patients' repository
            - Doctors' repository
            - Admins' repository
            - Appointments history
    - Viewed Pages
            - Personal Profile Page
            - All admins Page
            - All doctors Page
            - All patients Page
            - All Appointments Page
            - Appointment Booking Page
- Normal Admin Role

    - Specifications
        - A normal user with admin privileges
    - Allowed Actions
        - Update only its own personal data
        - Assign a user to patient or doctor role
        - Deactivate a patient or doctor
        - Book an appointment on behalf of a patient or doctor
        - Update Appointment Information
            - Update the appointment status (to Cancelled, Confirmed, Scheduled, Completed, in progress) based on the pre-set rules
            - Update the appointment date, time and description
    - Viewed Data
        - Patients' repository
        - Doctors' repository
        - Appointments history
    - Viewed Pages
        - Personal Profile Page
        - All doctors Page
        - All patients Page
        - All Appointments Page
        - Appointment Booking Page
- Patient Role
    - Specifications
        - A normal user with patient privileges
    - Allowed Actions
        - Update only its own personal data
        - Book an appointment on behalf of a patient or doctor
        - Update Appointment Information
            - Cancel Appointment
            - Update the appointment date, time and description
    - Viewed Data
        - Their own appointments
        - Their own personal info
    - Viewed Pages
        - Personal Profile Page
        - Their own Appointments Page
        - Appointment Booking Page
- Doctor Role
    - Specifications
        - A normal user with doctor privileges
    - Allowed Actions
        - Update only its own personal data
        - Book an appointment (only with them) on behalf of a patient
        - Update Appointment Information
            - Update the appointment status (to Cancelled, Confirmed, Scheduled, Completed, in progress) based on the pre-set rules

        - Update the appointment date, time and description
    - Viewed Data
        - Their own appointments
        - Their own personal info
    - Viewed Pages
        - Personal Profile Page
        - Their own Appointments Page
        - Appointment Booking Page
- User Role
    - Specifications
        - A freshly registered user
    - Allowed Actions
        - Update only its own personal data
    - Viewed Data
        - Their own personal info
    - Viewed Pages
        - Personal Profile Page

More implementation role implementation details will be given throughout this report.

1.2. Check Role Validity for Resource Access

Additional backend role check with sensitive API endpoint calls before starting the request processing. As visible below, the code checks for admin privileges through the get_current_admin function.

```python
@router.post("/getNonAssignedUsers/")
def get_non_assigned_users( db: Session = Depends(get_db), current_user: User = Depends(get_current_admin)):
    user_data = db.query(User).filter(User.is_valid == 1, User.role_hash == user_hash).all()
    result = []
    if not user_data:
        return result
    else:
```

In this function, we compare the current user hash to the known admin_hash. If different, then an error is raised. This means the user is not an admin.

```python
def get_current_admin(current_user: User = Depends(get_current_user)):
    if current_user.role_hash != admin_hash:
        raise HTTPException(status_code=403, detail=admin_privileges)
    return current_user
```

Since roles are encrypted within the database using the Fernet algorithm which uses random initialization vectors (IV), the same role encrypted will result in different cipher texts; hence the need for a hash value to recognize hash role values. Hashing is done through the hash_lookup function using the sha256 algorithm. This function can be found within the same file encryption.py.

```python
def hash_lookup(text: str) -> str:      # to check for similar values
    return sha256(text.encode()).hexdigest()

patient_hash = hash_lookup("patient")
doctor_hash = hash_lookup("doctor")
admin_hash = hash_lookup("admin")
user_hash = hash_lookup("user")
```

1.3. Scheduled Access Validity Check

To avoid expired users still gaining access to our system and misusing it. We have set up a daily database check, in which every day at midnight, the function deactivate_expired_users is called.

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    scheduler.add_job(
        deactivate_expired_users,     # remove users with expired status and set appointments to cancelled
        CronTrigger(hour = 0, minute = 0), #every day at Midnight
        id="deactivate_expired_users",
        replace_existing=True
    )
```

In fact, this function goes through all database tables, and checks in case the expiry_date is set if it date arrived or not. If yes, the deactivate_user function is called. Otherwise, nothing happens.

```python
# remove users with expired status and set appointments to cancelled
Tabnine | Edit | Test | Explain | Document
def deactivate_expired_users(current_admin: User = Depends(get_default_admin)):
    db = SessionLocal()
    date_today = datetime.now()
    users = db.query(User).filter(User.is_valid == 1).all()
    for user in users:
        patient_data = None
        doctor_data = None
        admin_data = None
        if user.role_hash == patient_hash:
            patient_data = db.query(Patient).filter(Patient.user_id == user.user_id, Patient.status_expiry <= date_today).first()
        elif user.role_hash == doctor_hash:
            doctor_data = db.query(Doctor).filter(Doctor.user_id == user.user_id, Doctor.status_expiry <= date_today).first()
        elif user.role_hash == admin_hash:
            admin_data = db.query(Admin).filter(Admin.user_id == user.user_id, Admin.status_expiry <= date_today).first()
        if patient_data or  doctor_data or admin_data:
            deactivate_user(user.user_id, db)
    db.close()
```

deactivate_user function on the other hand, is a general function that is called to deactivate a user in general. Based on the role, this function set the status (is_doctor, is_admin, is_patient) to invalid. And cancels future appointments with the respective doctor or patient if user is a doctor or patient.

```python
# Deactivate a user
Tabnine | Edit | Test | Explain | Document
@router.delete("/deactivate/{user_id}")
def deactivate_user(
    user_id: int,
    db: Session = Depends(get_db),
    current_admin: User = Depends(get_current_admin)):
    if user_id == 1:
        raise HTTPException(status_code=status.HTTP_403_FORBIDDEN, detail= default_admin_update )
    user =  get_valid_user(user_id, db)

    # set user_status to invalid and status_expiry date:
    if user.role_hash == patient_hash:
        search_db = db.query(Patient).filter(Patient.user_id == user_id).all()
        for current_user in search_db:
            if current_user.is_patient == True:
                current_user.is_patient = False
                if not current_user.status_expiry:
                    current_user.status_expiry = datetime.now(timezone.utc)
                db.add(current_user)

    elif user.role_hash == doctor_hash:
        search_db = db.query(Doctor).filter(Doctor.user_id == user_id).all()
        for current_user in search_db:
            if current_user.is_doctor == True:
                current_user.is_doctor = False
                if not current_user.status_expiry:
                    current_user.status_expiry = datetime.now(timezone.utc)
                db.add(current_user)

    elif user.role_hash== admin_hash:
        search_db = db.query(Admin).filter(Admin.user_id == user_id).all()
        for current_user in search_db:
            if current_user.is_admin == True:
                current_user.is_admin = False
                if not current_user.status_expiry:
                    current_user.status_expiry = datetime.now(timezone.utc)
                db.add(current_user)

    # Set scheduled appointments to CANCELLED
    if user.role_hash == patient_hash or user.role_hash == doctor_hash:
        appointments = get_user_appointments_by_user_id(user_id, db)
        for appointment in appointments:
            if appointment.status!= "CANCELLED":
                deactivate_appointment(appointment.appointment_id, db)

    # deactivate entry in User table
    user.is_valid = False
    db.commit()
    return {"message": "User Deactivated Successfully"}
```

2. A02:2021-Cryptographic Failures

2.1. HTTPS usage

We have implemented HTTPS (Hypertext Transfer Protocol Secure) in both the front and backend code. HTTPS is a secure version of HTTP in which data is transferred in an encrypted format rather than plain text.

We have used the mkcert Tool to generate a self-signed certificate and private key which can be found in the certificate folder

To configure HTTPs:

2.1.1. In Angular:

We have updated the angular.json file. We've set the protocol SSL (Secure Sockets Layer) to true and added a path to our private key and self-generated certificate.

```
    },
    "development": {
      "buildTarget": "app:build:development",
      "ssl": true,
      "sslKey":"../Certificate/key.pem",
      "sslCert":"../Certificate/cert.pem"
    }
  },
  "defaultConfiguration": "development"

},
```

2.1.2. In FastAPI:

We have updated our main.py file

```
if __name__ == "__main__":
    uvicorn.run(
    app,
    host="0.0.0.0",
    port = 8432,
    ssl_keyfile="../../Certificate/key.pem",
    ssl_certfile="../../Certificate/cert.pem",
    lifespan="on",
)
```

We are using uvicorn which is the server running fast API and are allowing it to listen to all ports within the network (host = “0.0.0.0”), setting our backend port, we have chosen an arbitrary port 8432, and as in angular set up, we also specify certificate and private key files path.

2.2. Sensitive Data Encryption

We have encrypted sensitive end users’ information: Full Name, Email, Phone Number, Role, Appointment Description, username, and doctor specialty. For that, we’ve relied on Fernet which is a Symmetric Encryption Algorithm providing confidentiality, integrity, and authentication. The corresponding code can be found within the encryption.py file.

```
# Get the DB_SECRET_KEY from the environment
DB_SECRET_KEY = settings.DB_SECRET_KEY


hashed_key = sha256(DB_SECRET_KEY.encode()).digest() # Hash DB_SECRET_KEY  to produce a 32-byte Fernet encryption key
cipher = Fernet(base64.urlsafe_b64encode(hashed_key)) # Ensure the generated key is properly formatted

Tabnine | Edit | Test | Explain | Document
def encrypt(text: str) -> str:
    try:
        cipher_text = cipher.encrypt(text.encode()).decode()
        return cipher_text
    except Exception:
        raise HTTPException(status_code=500, detail= internal_error)


Tabnine | Edit | Test | Explain | Document
def decrypt(token: str) -> str:
    try:
        plain_text = cipher.decrypt(token.encode()).decode()
        return plain_text
    except Exception:
        raise HTTPException(status_code=500, detail= internal_error)
```

The code first gets the personalized key from the environment. The key is then hashed using the SHA-256 algorithm to ensure the generated key is 32 bytes. After that, the key is passed to the urlsafe_b64encode function, which ensures the key is properly formatted and is

compatible with Fernet expected format. Finally, the generated Fernet key called cipher in our code is used to call either encrypt or decrypt prebuilt functions.

2.3. Strong Password Hashing

We have used Bcrypt for one-way password hashing. Hence, even if an attacker gains access to our database they cannot recover the password. The corresponding code can be found within security.py.

```
# Hash password using bcrypt
Tabnine | Edit | Test | Explain | Document
def hash_password(password: str) -> str:
    return bcrypt.hashpw(password.encode("utf-8"), bcrypt.gensalt()).decode("utf-8")

# Verify password using bcrypt
Tabnine | Edit | Test | Explain | Document
def verify_password(plain_password: str, hashed_password: str) -> bool:
    return bcrypt.checkpw(plain_password.encode("utf-8"), hashed_password.encode("utf-8"))
```

For hash_password, the function uses, within hashpw pre-built function, encode to convert the password to utf-8-character format password and gensalt() pre-built function to generate a random salt.

For password verification, we use the checkpw pre-built function which uses both the stored hashed password, and the plain password encoded in utf-8 format.

3. A03:2021-Injection

3.1. Built-in Input Validation

In fact, FastAPI uses Pydantic which allows validating user input before processing it, in terms of expected type and format and constraint adherence.

```
@router.put("/adminUpdateAppointment/")
def admin_update_appointment(data: admin_appointment_update,
                             db: Session = Depends(get_db),
                             current_admin: User = Depends(get_current_admin)):
    appointment = db.query(Appointment).filter(Appointment.appointment_id == data.id).first()
    if not appointment:
        raise HTTPException(status_code=404, detail=appointment_not_found)

    if appointment.status =="COMPLETED" or appointment.status =="CANCELLED":
```

As an example, this function expects data to conform with the admin_appointment_update pydantic model.

Which was defined separately within the schemas folder.

```
class admin_appointment_update(BaseModel):
    id: int  #  appointment_id
    description : str
    date_time : datetime
    status : str
```

Hence, while calling this function, the client must pass exactly 4 variables with the naming and data type specified above.

4. A04:2021-Insecure Design

4.1. Threat Modeling

To do our Threat Identification and Response/ Mitigation, we have used the STRIDE technique.

Threat Category	Threats Example	Mitigation/ Response
Spoofing	Clickjacking attacks through iframe	x-frame-options header
Spoofing	Tab-nabbing attacks	Cross-Origin-Opener-Policy header
Tampering	Tampere JWT token data	A strong algorithm is used (H256) to sign and verify signatures
Tampering	MIME sniffing attacks	X-Content-Type-Options header
Tampering	Attacker enters unexpected data	Input validation: Validate all user inputs through Pydantic model
Tampering	Cross origin scripts used with no proper right	Cross-Origin-Embedder-Policy header
Tampering	Outdated appointment statuses	Scheduled Update Appointments Status check
Tampering	Data Inconsistency for expired users	Access Validity Check to deactivate expired user related records
Repudiation	-Can't revoke tokens - Attacker uses a still valid logged out user token	Token Blacklist Implentation
Repudiation	Once access is gained, the attacker has plenty of time to misuse the system	Token Expiry
Information Disclosure	The Attacker gains access to the database and extracts confidential user data.	Sensitive Data Encryption
Information Disclosure	Leaking User Data through Error messages	Generic messages are used for all raised errors
Information Disclosure	Data disclosed through cross-origin	Cross-Origin-Resource-Policy header
Information Disclosure	Sensitive data sent through HTTPS	HTTPS usage with HSTS header
Denial of Service	Attacker sending too many requests at the same time	Rate limiting this is not implemented, we aim to implement it in the improvement part
Elevation of Privileges	-User Accessing Restricted Endpoints -Users gets more information than needed	Role Based Access Control And JWT authentication
Elevation of Privileges	Restricted or expired users still using the system	Scheduled Access Validity Check
Elevation of Privileges	Usage of restricted browser APIs	Permissions-Policy header

4.2. Secure Design Patterns and Principles

4.2.1. Least Privileges and Separation of Duties

Design the system to give the various users only the access rights needed to perform their tasks. This was ensured with:

i. Role Based Access Control (RBAC)

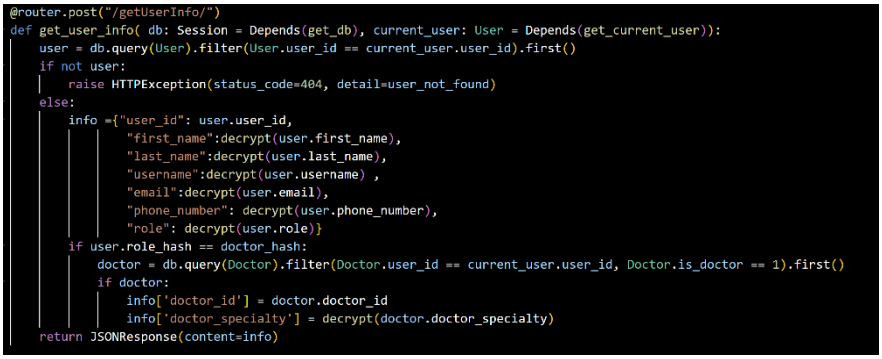
As explained earlier in detail, within the "A01:2021-Broken Access Control" part, we have designed our project with roles to restrict access and separate duties.

ii. Restrict Client's Side Data

All our back-end functions are designed similarly. Only necessary data and data related to these specific user rights are sent to the front end.

In fact, in this function, we are only fetching the user's specific data in the database through the filter option where we specify that we are looking for a database row where the user_id is the same as our current user's.

And from this data, we only send the data our frontend needs.



```
@router.post("/getUserInfo/")
def get_user_info( db: Session = Depends(get_db), current_user: User = Depends(get_current_user)):
    user = db.query(User).filter(User.user_id == current_user.user_id).first()
    if not user:
        raise HTTPException(status_code=404, detail=user_not_found)
    else:
        info ={"user_id": user.user_id,
               "first_name":decrypt(user.first_name),
               "last_name":decrypt(user.last_name),
               "username":decrypt(user.username) ,
               "email":decrypt(user.email),
               "phone_number": decrypt(user.phone_number),
               "role": decrypt(user.role)}
        if user.role_hash == doctor_hash:
            doctor = db.query(Doctor).filter(Doctor.user_id == current_user.user_id, Doctor.is_doctor == 1).first()
            if doctor:
                info['doctor_id'] = doctor.doctor_id
                info['doctor_specialty'] = decrypt(doctor.doctor_specialty)
    return JSONResponse(content=info)
```

iii. Endpoints Protection

Access rights are checked with each endpoint call; ensuring only registered users with necessary permission privileges are getting back backend responses.

For this, 2 functions were created within the utils.py file.

get_current_user => to check if the requester is a user regardless of their role

get_current_admin => to check if the requester is an admin

get_current_user function is the parent function; it checks first if the user is logged out, decodes the token, and extracts from it the user ID and token expiry date. For each of the extracted values, the code checks its validity. For user ID, check for a valid user in the DB with the same ID. Token expiry is compared with the actual date at the moment. Based on these criteria, an exception is raised if one of them is not fulfilled.

```python
async def get_current_user(token: str = Depends(oauth2_scheme), db: Session = Depends(get_db)):
    credentials_exception = HTTPException(
        status_code=status.HTTP_401_UNAUTHORIZED,
        detail="Could not validate credentials",
        headers={"WWW-Authenticate": "Bearer"},
    )
    # verify expired session token
    try:
        if is_logged_out(token, db):
            raise HTTPException(status_code=400, detail=authentication_error)
        payload = jwt.decode(token, settings.JWT_SECRET_KEY, algorithms=[settings.ALGORITHM])
        user_id = payload.get("sub")
        if not user_id:
            raise credentials_exception
        token_exp = payload.get("exp")
        if datetime.utcnow().timestamp() > token_exp:
            raise HTTPException(
                status_code=status.HTTP_401_UNAUTHORIZED,
                detail="Expired Session",
                headers={"WWW-Authenticate": "Bearer"},
            )
        token_data = TokenData(user_id=user_id)
    except JWTError:
        raise credentials_exception

    user = db.query(User).filter(User.user_id==token_data.user_id).first()

    if user is None:
        raise credentials_exception
    return user
```

Get_current_admin relies on get_current_user to check the validity of the user, plus checks the user's role if it is set to admin or not. As explained earlier in the "A01:2021-Broken Access Control" part, the check is done through the already computed and set admin hash value.

```python
def get_current_admin(current_user: User = Depends(get_current_user)):
    if current_user.role_hash != admin_hash:
        raise HTTPException(status_code=403, detail=admin_privileges)
    return current_user
```

iv. User Interface display based on roles

The front end reinforces this principle by restricting access to pages based on roles. In fact, as seen below, the header differs from one role to another, hence page access

```
frontend > src > app > components > header > header.component.html > nav.navbar.navbar-expand-lg.bg-body-tertiary > div.container-fluid > div#navbarNav.colla
  3   <nav class="navbar navbar-expand-lg bg-body-tertiary">
  4     <div class="container-fluid">
  9       <div class="collapse navbar-collapse" id="navbarNav">
 10         <ul class="navbar-nav mx-auto">
 13           </li>
 14           <li class="nav-item dropdown"  *ngIf="isAdmin">
 15             <a class="nav-link dropdown-toggle" id="navbarDropdown" role="button" aria-expanded="false">
 16               Appointments
 17             </a>
 18             <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
 19               <li><a class="dropdown-item" routerLink="admin_new_appointment">New Appointment</a></li>
 20               <li><a class="dropdown-item" routerLink = "/Appointments_Admin_View">Appointment List</a></li>
 21             </ul>
 22           </li>
 23           <li class="nav-item"  *ngIf="isAdmin">
 24             <a class="nav-link" routerLink = "/Patients_Admin_View">Patients</a>
 25
 26           </li>
 27           <li class="nav-item"  *ngIf="isAdmin">
 28             <a class="nav-link" routerLink="/Doctors_Admin_View">Doctors</a>
 29           </li>
 30           <li class="nav-item"  *ngIf="isDefaultAdmin">
 31             <a class="nav-link" routerLink="/Admins_Default_Admin_View">Admins</a>
 32           </li>
 33
 34           <li class="nav-item dropdown"  *ngIf="isDoctor">
 35             <a class="nav-link dropdown-toggle" id="navbarDropdown" role="button" aria-expanded="false">
 36               Appointments
 37             </a>
 38             <ul class="dropdown-menu" aria-labelledby="navbarDropdown">
 39               <li><a class="dropdown-item" routerLink="doctor_new_appointment">New Appointment</a></li>
```

Also, in our code logic, the allowed actions and views differ from one role to another; hence the use of different components. Based on the role, the user is directed to the expected-to-be-viewed page.

```
async assignUser(){
  if (this.title =="appointments"){
    this.configService.getUserRole().subscribe(
      (response:any) => {
        const userRole = response;
        if (userRole =="patient"){
          this.router.navigateByUrl("/patient_new_appointment");
        }
        else if (userRole =="doctor"){
          this.router.navigateByUrl("/doctor_new_appointment");
        }
        else if (userRole =="admin"){
          this.router.navigateByUrl("/admin_new_appointment");
        }
      },
      (error:any) => {
        console.error('Error fetching data:', error);
      }
    );
  }
  else{
```

4.2.2. The principle of Defense-in-Depth

i. HTTPS and HSTS:

To ensure we have multiple layers of control, we are using HTTPS merged with HTTP Strict Transport Security (HSTS). And for this, we have set the following header in our angular code, server.ts file.

This header “strict-transport-security", tells browsers to always use HTTPS when sending requests to this domain. We have set the max age to 2 years and set the rule also for subdomains not only main domains.

```
function secureHeaders(res: express.Response){
  res.setHeader('X-Frame-Options', 'SAMEORIGIN');
  res.setHeader('X-Content-Type-Options', 'nosniff');
  res.setHeader("Cross-Origin-Resource-Policy", "same-origin");
  res.setHeader("Cross-Origin-Embedder-Policy", "require-corp");
  res.setHeader("Cross-Origin-Opener-Policy", "same-origin");
  res.setHeader("Permissions-Policy", "geolocation=(), camera=(), microphone=(), fullscreen=(self)");
  res.setHeader("Strict-Transport-Security", "max-age=63072000; includeSubDomains; preload");
  res.removeHeader("X-Powered-By");
}
Tabnine | Edit | Test | Explain | Document
```

4.2.3. The principle of Zero Trust

i. Never Trust Client Side:

By design, we never trust our front end, we always double-check the access rights, type of data sent, format of data, and number of arguments. As explained earlier, in the “Endpoints Protection” part and “Built-in Input Validation” part.

ii. Continuous login Status Check:

For that, we have set a short token validity duration of 30 mins only as specified in the settings file.

```python
BASE_DIR = os.path.dirname(os.path.dirname(os.path.dirname(os.path.abspath(__file__)))) # return backend folder
ENV_FILE = os.path.join(BASE_DIR, ".env")

class Settings(BaseSettings):
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30
    ALGORITHM: str = 'HS256'
    JWT_SECRET_KEY: str
    DB_SECRET_KEY: str
    env: str

    class Config:
        env_file = ENV_FILE
        env_file_encoding = "utf-8"

settings = Settings()
```

Which is continuously checked whenever an endpoint is called through the get_current_user and get_current_admin functions. These functions compare the extracted token expiry date to the current date and raise an error if expires. The check is also done, through is_log_out, which checks if the token is expired or is set in the blacklist table.

iii. Token Blacklist Implementation:

Because we never trust our system’s users, a token blacklist implementation was set.

In fact, whenever a user logs out, its corresponding token is added to the blacklist table.

```python
@router.post("/logout/")
def logout(
    token: str = Depends(oauth2_scheme),
    current_user: User = Depends(get_current_user),
    db: Session = Depends(get_db)
):
    decoded_token = verify_token(token, db)
    if decoded_token:
        # Blacklist token
        blacklist_db = TokenBlacklist(access_token = token, expired_at = datetime.utcnow() )
        db.add(blacklist_db)
        db.commit()
        db.refresh(blacklist_db)
        return {"msg": "DONE"}
```

For verify_token, it only checks if the token is already added to the blacklist or not. If not, it decodes the token and returns it.

```python
def verify_token(token : str, db: Session = Depends(get_db)):
    #search for token in the blacklist_token table
    blacklist_token = db.query(TokenBlacklist).filter(TokenBlacklist.access_token == token).first()
    if blacklist_token:
        raise HTTPException(status_code=401, detail=already_logged_out)
    return decode_token(token)
```

For the decode_token function, it only calls the JWT decode built-in function to decode the token and raises an error in case of issues.

This technique has many advantages. In fact, since JWT is stateless, tokens remain valid until they expire. The attacker can make use of these non-expired tokens to misuse the system. However, by blacklisting a token, once logged out, the token is no longer valid even though it still didn't expire.

Additionally, this can allow system administrators to revoke a token whenever it is needed. Mainly, when encountering suspicious user behavior, they can immediately interrupt their system's right access.

5. A05:2021-Security Misconfiguration

5.1. Enabled production mode, in both our frontend and backend codes

5.1.1. In Angular:

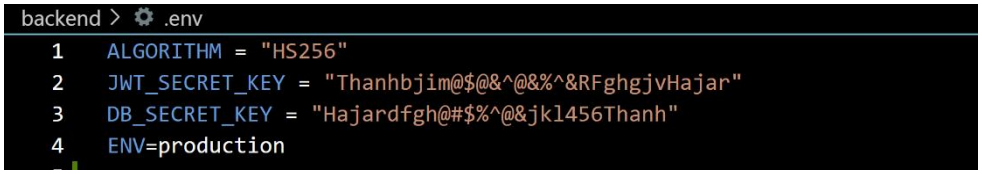
We have built our project using the following command "ng build --configuration production"

We have updated our code more precisely within the server.ts file and added a call to enableProdMode() as shown in the picture below to disable debugging tools in production

5.1.2. In fastAPI:

We have blocked access in production to API documentation endpoints

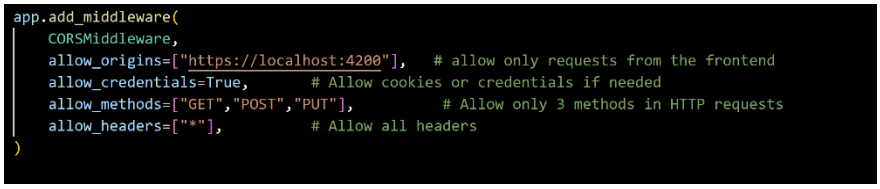
To know if we are in production mode or not, the code gets the value of ENV within .env file



5.2. We have set CORS policies

5.2.1. In fastAPI:

We have restricted origins and methods, as seen below, only requests from our frontend are allowed with only three possible methods, the one used by our API (GET, POST and PUT).



5.3. Data Consistency and Integrity

To avoid attackers benefiting from bugs and system misconfigurations to violate our already set rules, in addition to the daily scheduler tackled in detail within the “A01:2021-Broken Access Control” part, we have added a regular appointment status check which happens every 30 minutes, to keep checking appointments status and update them accordingly.

```python
@asynccontextmanager
async def lifespan(app: FastAPI):
    scheduler.add_job(
        deactivate_expired_users,     # remove users with expired status and set appointments to cancelled
        CronTrigger(hour = 0, minute = 0), #every day at Midnight
        id="deactivate_expired_users",
        replace_existing=True
    )
    scheduler.add_job(
        check_appointments,
        'interval', minutes= 30,  #check appointments' status each 30 mins
        id="check_appointments",
        replace_existing=True)
    scheduler.start()
    try:
        yield
    finally:
        scheduler.shutdown()
```

In fact, the check_appointments function sets appointments within 1 hour from the starting time to IN PROGRESS status. After that, it sets the appointment to COMPLETED. Otherwise, it does nothing.

```python
def check_appointments():
    db = SessionLocal()
    appointments = db.query(Appointment).filter(Appointment.status != "COMPLETED", Appointment.status != "CANCELLED").all()
    for appointment in appointments:
        now = datetime.now()
        if appointment.date_time <= now < appointment.date_time + timedelta(hours=1): #within 1 hour of appointment starting time
            appointment.status = "IN PROGRESS"
        elif now >= appointment.date_time + timedelta(hours=1): # more than 1 hour since the appointment starting time
            appointment.status = "COMPLETED"
        #else: future date
        db.commit()
        db.refresh(appointment)
    db.close()
```

6. A06:2021-Vulnerable and Outdated Components

6.1. Dependency and Vulnerability Check

As it will be mentioned in the security testing part, we have used static scanning and CI/CD pipeline to test our code vulnerabilities and update the code accordingly.

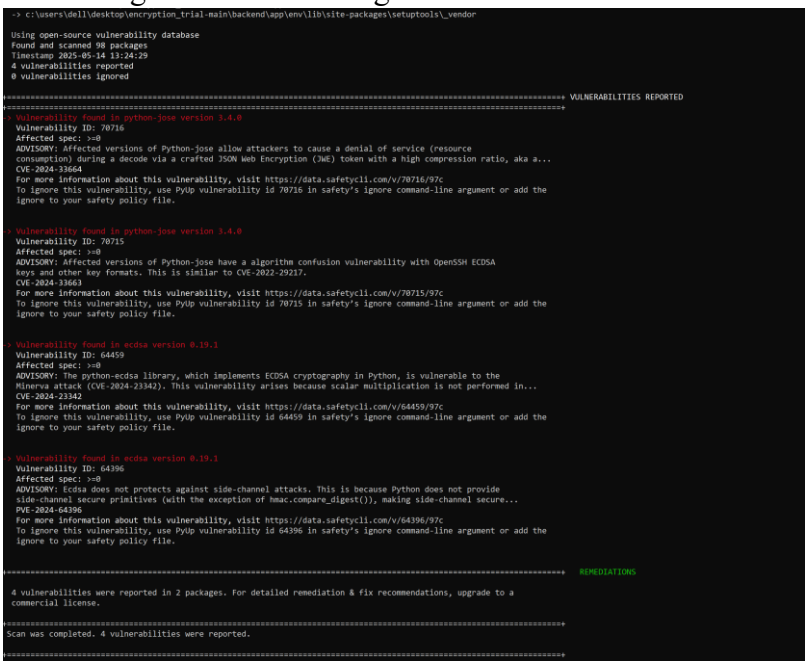
Moreover, we have used two different libraries in the backend pip-audit and safety

- pip-audit:
  - Showed no known vulnerability

```
(env) C:\Users\DELL\Desktop\encryption_trial-main\backend\app>pip-audit
No known vulnerabilities found
```

- safety:
  - 1st trial:
    - Showed 5 vulnerabilities related to:
      - Python-jose version
      - Pip version
      - Ecdsa version

- 2nd trial:
  - After installing the latest versions, two library vulnerabilities were still left:
    - Python-jose version
    - Ecdsa version
  - However, we have ignored these vulnerabilities since our code logic is not affected
    - We are only accepting signed JWT tokens using HS256 algorithm
    - We are not using ECDSA-based algorithms



7. A07:2021-Identification and Authentication Failures

7.1. Enforce strong password policy

We use regex to check the user's password, if the password does not meet any criteria below, the backend will return False and send a message "Weak Password!" to the frontend.

Strong password criteria:

- Minimum length: 8 characters
- Has at least 1 uppercase
- Has at least 1 lowercase
- Has at least 1 digit
- Has at least 1 special character (for example, !@#$%^&*()|<>)

```python
def is_strong_password(password: str):
    # Password should has criteria such as:
    # Minimum length: 8 characters
    # At least one uppercase
    # At least one lowercase
    # At least one digit
    # At least one special character
    if len(password) < 8 or
        not re.search(r'[A-Z]', password) or
        not re.search(r'[a-z]', password) or
        not re.search(r'\d', password) or
        not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        return False
    return True
```

The password stored in our database is encrypted with Bcrypt as explained earlier.

7.2. OAuth2 and JWT Tokens Usage

We use OAuth2 to request a username and password from the user, then normally compare these credentials with data saved in our database.

```python
# User login and return token
@router.post("/login/")
async def login_for_access_token(
    form_data: OAuth2PasswordRequestForm = Depends(),
    db: Session = Depends(get_db)
    ) -> Token:
    user = authenticate_user(form_data.username, form_data.password, db)
    if not user:
        raise HTTPException(
            status_code=status.HTTP_401_UNAUTHORIZED,
            detail="Incorrect Username or Password",
            headers={"WWW-Authenticate": "Bearer"},
        )
    access_token_expires = timedelta(minutes=settings.ACCESS_TOKEN_EXPIRE_MINUTES)
    access_token = create_access_token(
        data={"sub": str(user.user_id)}, expires_delta=access_token_expires
    )
    return Token(access_token=access_token, token_type="bearer")
```

After logging in with a valid username and password, JWT tokens are created and used to manage the session with an expiration time of 30 minutes.

```python
# create Json Web Token (JWT)
def create_access_token(data: dict, expires_delta: timedelta | None = None) -> str:
    to_encode = data.copy()

    # set the expiration date: expires_delta: in case it is given otherwise generate a new one based on
    # ACCESS_TOKEN_EXPIRE_MINUTES settings
    expire = datetime.now(timezone.utc) + (expires_delta or timedelta(minutes=ACCESS_TOKEN_EXPIRE_MINUTES))

    #add the new key-value exp
    to_encode.update({"exp": expire})

    return jwt.encode(to_encode, JWT_SECRET_KEY, algorithm=ALGORITHM)
```

7.3. Tokens validation

As explained above, we not only manage tokens with expired time but also blacklist tokens for invalidated tokens during logout.

B. SANS 25

1. Buffer Overflow (CWE-119, CWE-120, CWE-121, CWE-122, CWE-124)

We are implementing our backend using Python. Python manages memory automatically with built-in safety checks which helps prevent these buffer overflow CWEs

2. Information exposure (CWE-200)

We use general error messages not to expose important information

All error messages are aggregated in the message.py file for easy management. Each message only shows general information and does not reveal any specific data.

```python
doctor_not_found = "Doctor Not Found"
admin_not_found = "Admin Not Found"
patient_not_found = "Patient Not Found"
user_not_found = "User Not Found"
appointment_not_found = "Appointment Not Found"
non_updatable_appointment = "Non Updatable Appointment"

default_admin_update = "Default Admin Cannot Be Updated"
default_admin_missing = "Default Admin is Missing"
default_admin_privileges = "Default Administrator Privileges are Needed"
admin_privileges = "Administrator Privileges are Needed"
doctor_privileges = "Doctor Privileges are Needed"
patient_privileges = "Patient Privileges are Required"
general_privileges_update = "You Are Not Allowed to Update This Record"
general_privileges_getInfo = "You Are Not Allowed to Get This Information"

invalid_role = "Invalid Role"
invalid_date = "Invalid Date"
invalid_name = "Invalid Name"
invalid_email= "Invalid Email"
invalid_phone_number = "Invalid Phone Number"
invalid_token = "Invalid Token"
invalid_doctor_specialty = "Invalid Doctor Specialty"
invalid_status = "Invalid Appointment Status"
invalid_chosen_status = "Cannot Be Marked For The Moment as "
expired_token = "Token has expired"
```

- We use security headers in both the front and backend
  - ♦ Anti-clickjacking Header protects against clickjacking attacks
  - ♦ Permissions Policy Header controls the browser's features access such as camera and geolocation
  - ♦ X-Content-Type-Options Header protects against browsers' MIME-sniffing content types and some XSS attacks
  - ♦ Site Isolation Against Spectre Vulnerability limits data leakage across sites
  - ♦ Content Security Policy (CSP) Header detects and protects against some attacks such as Cross-Site Scripting (XSS) and data injection
  - ♦ Caching: Controls how responses are cached

- "X-Powered-By" (Removed Header) prevents the server from leaking information via "X-Powered-By" HTTP response headers

These headers reduce the amount and sensitivity of information that could be exposed to users, and harden the application against data leaks.

- Backend

```python
class secureHeader(BaseHTTPMiddleware):
    Tabnine | Edit | Test | Explain | Document
    async def dispatch(self, request: Request, call_next):
        response: Response = await call_next(request)
        if request.url.path == "/docs":  # to allow only FASTAPI UI
            return response
        response.headers["X-Frame-Options"] = "SAMEORIGIN" # Missing Anti-clickjacking Header
        response.headers["Content-Security-Policy"] = "default-src 'self';" "script-src 'self';" "font-src 'self'; " "connect-src 'self';" "frame-ancestors 'none';" "form-action 'self';" #  Content Security
        response.headers["Cross-Origin-Resource-Policy"] = "same-origin" #  Insufficient Site Isolation Against Spectre Vulnerability
        response.headers["Cross-Origin-Embedder-Policy"] = "require-corp" #     Insufficient Site Isolation Against Spectre Vulnerability
        response.headers["Cross-Origin-Opener-Policy"] = "same-origin" #    Insufficient Site Isolation Against Spectre Vulnerability
        response.headers["Permissions-Policy"] = "geolocation=(), camera=(), microphone=(), fullscreen=(self)" # Permissions Policy Header Not Set
        response.headers["X-Content-Type-Options"] = "nosniff" # X-Content-Type-Options Header Missing
        del response.headers["X-Powered-By"] # Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)

        # disable caching
        response.headers["Cache-Control"] = "no-cache, no-store, must-revalidate, private"
        response.headers["Pragma"] = "no-cache"
        response.headers["Expires"] = "0"
        return response


app.add_middleware(secureHeader)
```

- Frontend

```javascript
function secureHeaders(res: express.Response){
 res.setHeader('X-Frame-Options', 'SAMEORIGIN');
 res.setHeader('X-Content-Type-Options', 'nosniff');

 res.setHeader("Cross-Origin-Resource-Policy", "same-origin");
 res.setHeader("Cross-Origin-Embedder-Policy", "require-corp");
 res.setHeader("Cross-Origin-Opener-Policy", "same-origin");
 res.setHeader("Permissions-Policy", "geolocation=(), camera=(), microphone=(), fullscreen=(self)");
 res.setHeader("Strict-Transport-Security", "max-age=63072000; includeSubDomains; preload");

 res.removeHeader("X-Powered-By");
}
```

We check authority each time an API is requested (if needed) so that unauthorized action cannot be implemented

3. Use of Hard-coded Credentials (CWE-79)

Secrets are stored securely in “environment” variables

We stored secrets like JWT_SECRET_KEY and DB_SECRET_KEY as environment parameters in .env file separately.

```
ALGORITHM = "HS256"
JWT_SECRET_KEY = "Thanhbjim@$@&^@&%^&RFghgjvHajar"
DB_SECRET_KEY = "Hajardfgh@#$%^@&jkl456Thanh"
ENV=production
```

- These secrets are then parsed to the application in config.py file

```python
class Settings(BaseSettings):
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30
    ALGORITHM: str = 'HS256'
    JWT_SECRET_KEY: str
    DB_SECRET_KEY: str
    env: str

    class Config:
        env_file = ENV_FILE
        env_file_encoding = "utf-8"
```

4. Cross-Site Scripting (CWE-79)

In the front end, we use the framework Angular, which has built-in protections against XSS

In the backend, we use Pydantic validation to sanitize inputs, as explained above

5. Cross-Site Request Forgery - CSRF (CWE-352)

We use JWT in Authorization header. JWT is stored in localStorage and sent in Authorization: Bearer <token> header on each API request to prevent CSRF using Interceptor in Angular.

```
export const tokenInterceptor: HttpInterceptorFn = (req, next) => {
  const router = inject(Router);
  const authService = inject(AuthService);
  const token = localStorage.getItem("token");
  const newReq = req.clone({
    setHeaders:{
      Authorization: `Bearer ${token}`
    }
  })
```

By using this method, the browser does not automatically send the token on every HTTP request (like cookies). Instead, Interceptor explicitly adds the JWT into the Authorization Header for API calls, which helps prevent CSRF since the attackers cannot make a forged request with a valid JWT Authorization header on behalf of the user.

IV. Security Testing

A. Manual Security Testing

To test our app, we first started with manual testing, in which we tested basic features and code behavior in case of unexpected inputs or unusual conditions. While testing, we kept updating our code, accordingly, adding new security tests, raising errors, and restricting actions.

As an example, an admin used to be able to set an appointment to COMPLETED status even though the appointment date still hadn't arrived. Hence, we have to restrict this action and allow it only if the appointment date has passed.

```
    if data.status:
        if data.status.upper() not in allowed_status:
            raise HTTPException(status_code=404, detail= invalid_status)

        if data.status.upper() == "COMPLETED" and data.date_time > datetime.now(timezone.utc):
            raise HTTPException(status_code=404, detail= invalid_chosen_status + data.status.upper() )

        if data.status.upper() == "IN PROGRESS" and data.date_time != datetime.now(timezone.utc):
            raise HTTPException(status_code=404, detail= invalid_chosen_status + data.status.upper())

        appointment.status = data.status.upper()
    db.commit()
```

Additionally, previously we allowed user role change, which might result in users having access to data and privileges more than necessary. To avoid this, a role is set once and can't be changed. Hence, by trying to update the user role, an exception will be raised.

```python
@router.put("/updateRole/", response_model=UserUpdate)
def update_user_role(
    role_update: RoleUpdate,
    db: Session = Depends(get_db),
    current_admin: User = Depends(get_current_admin)):
    if role_update.user_id == 1:
        raise HTTPException(status_code=404, detail=default_admin_update)
    roles = ['admin', 'doctor', 'patient', 'user']
    role_update.new_role = role_update.new_role.lower()
    if role_update.new_role not in roles:
        raise HTTPException(status_code=404, detail = invalid_role)

    db_user =  get_valid_user(role_update.user_id, db)
    if db_user.role_hash != user_hash and db_user.role_hash != hash_lookup(role_update.new_role):
        raise HTTPException(status_code=404, detail= user_role_change)
```

After that, we moved to testing our code against known attack penetrations and started fixing them. An example of this would be the clickjacking attack. This is not a complete clickjacking attack; the attacker still needs to make the iframe invisible or disguised as another element, but still, we have used it to test our endpoints.

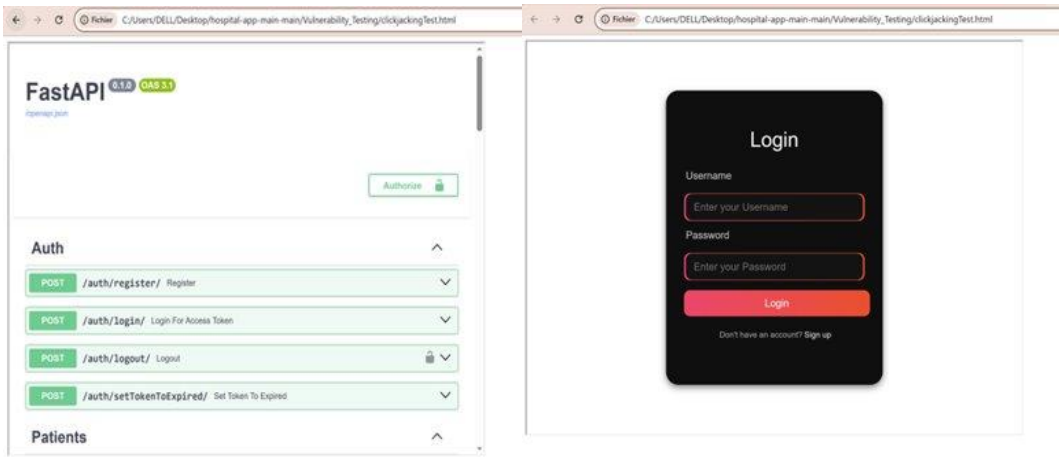
```html
<!DOCTYPE html>
<html>
<head>
  <title>Clickjacking Vulnerability Test</title>
</head>
<body>

  <!-- ***************** To Test Backend  ***************** -->
  <iframe src="https://localhost:8432/docs" width="800" height="600"></iframe>


  <!-- ***************** To Test Frontend  ***************** -->
  <iframe src="https://localhost:4200" width="800" height="600"></iframe>

</body>
</html>
```

As a result, both frontend and backend endpoints were accessible. As visible in the pictures below.



To avoid this, as stated in our earlier part “Information exposure (CWE-200)”, we have added the X-FRAME-OPTIONS header in both the front and back end and set it to the same origin.

As a result, the system is no longer accessible through iframe.

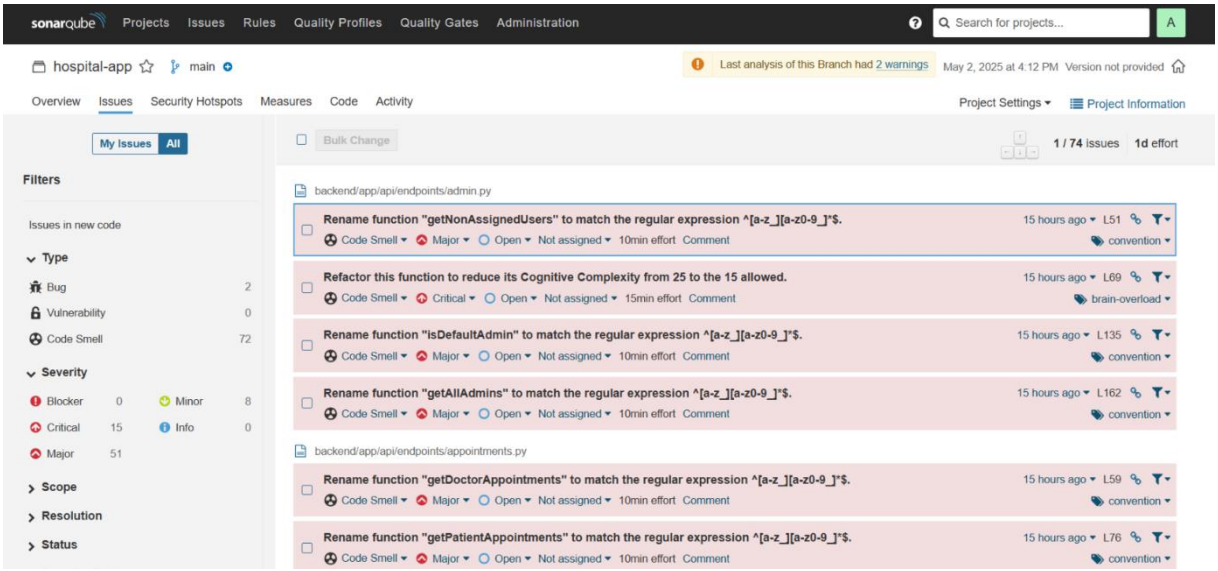
B. Security Testing in the CI/CD Pipeline

Please check the full reports in [github]

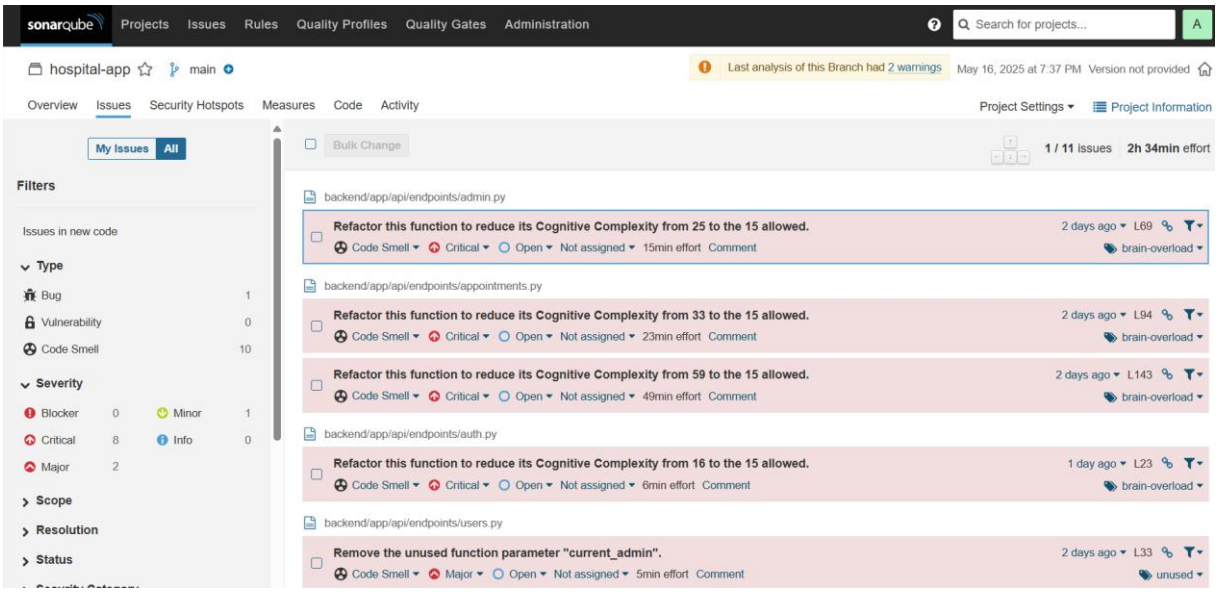
1. Static Application Security Testing (SAST)

SonarQube report: In total 74 issues

- Functions' name not match regular expression
- High Cognitive Complexity
- Duplicated code
- Variable is named similar to builtin
- Lack of exception
- Commented code should be removed
- Unused function parameter
- Unexpected empty source



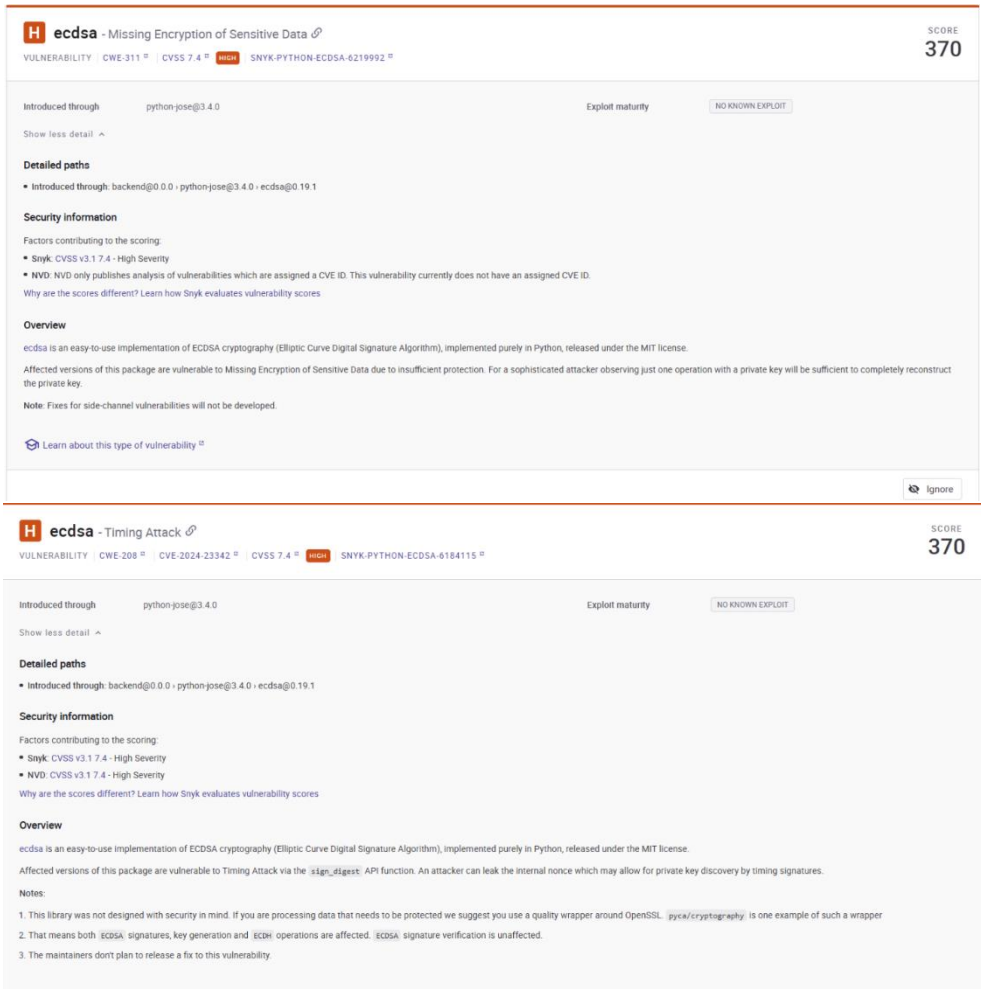
Fixed: We have tried to fix most of the issues in SonarQube following its instruction and reduced the issues to 11. These issues left are about High Cognitive Complexity.



2. Software Composition Analysis (SCA)

Snyk report:

- Frontend has 0 issues
- Backend has 2 issues related to python-jose@3.4.0



3. File System and Container Scanning

Trivy Container Scan:

- Backend:
  - only found vulnerabilities related to Debian, no misconfigurations found
- Frontend:
  - Debian: has vulnerabilities, no misconfigurations
  - Gobinary: 1 vulnerability for *stdlib*
  - Node-pkg: 2 vulnerabilities for *http-proxy-middleware*, 3 vulnerabilities for *vite*.

4. Dynamic Application Security Testing (DAST)

ZAP report:

Name	Risk Level	Number of Instances
CSP: Failure to Define Directive with No Fallback	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	3
Missing Anti-clickjacking Header	Medium	3
Sub Resource Integrity Attribute Missing	Medium	7
Dangerous JS Functions	Low	1
Insufficient Site Isolation Against Spectre Vulnerability	Low	12
Permissions Policy Header Not Set	Low	10
Server Leaks Information via "X-Powered-By" HTTP Response Header Field(s)	Low	6
X-Content-Type-Options Header Missing	Low	9
Authentication Request Identified	Informational	1
Information Disclosure - Sensitive Information in URL	Informational	4
Information Disclosure - Suspicious Comments	Informational	8
Modern Web Application	Informational	1
Non-Storable Content	Informational	2
Storable and Cacheable Content	Informational	5
Storable but Non-Cacheable Content	Informational	5

Fixed: We have added security headers (showed in III) based on this report in both frontend and backend, changed GET requests including sensitive data to POST request to avoid data exposure.

Name	Risk Level	Number of Instances
CSP: Failure to Define Directive with No Fallback	Medium	2
Content Security Policy (CSP) Header Not Set	Medium	3
Sub Resource Integrity Attribute Missing	Medium	1
Dangerous JS Functions	Low	1
Insufficient Site Isolation Against Spectre Vulnerability	Low	8
Permissions Policy Header Not Set	Low	9
Strict-Transport-Security Header Not Set	Low	8
X-Content-Type-Options Header Missing	Low	8
Authentication Request Identified	Informational	1
Information Disclosure - Sensitive Information in URL	Informational	4
Information Disclosure - Suspicious Comments	Informational	7
Modern Web Application	Informational	1
Non-Storable Content	Informational	6
Storable but Non-Cacheable Content	Informational	5

The issues left are with files created in memory with ng serve in development mode.

V. Security Issue or Vulnerability

- Backend:
  - High Cognitive Complexity

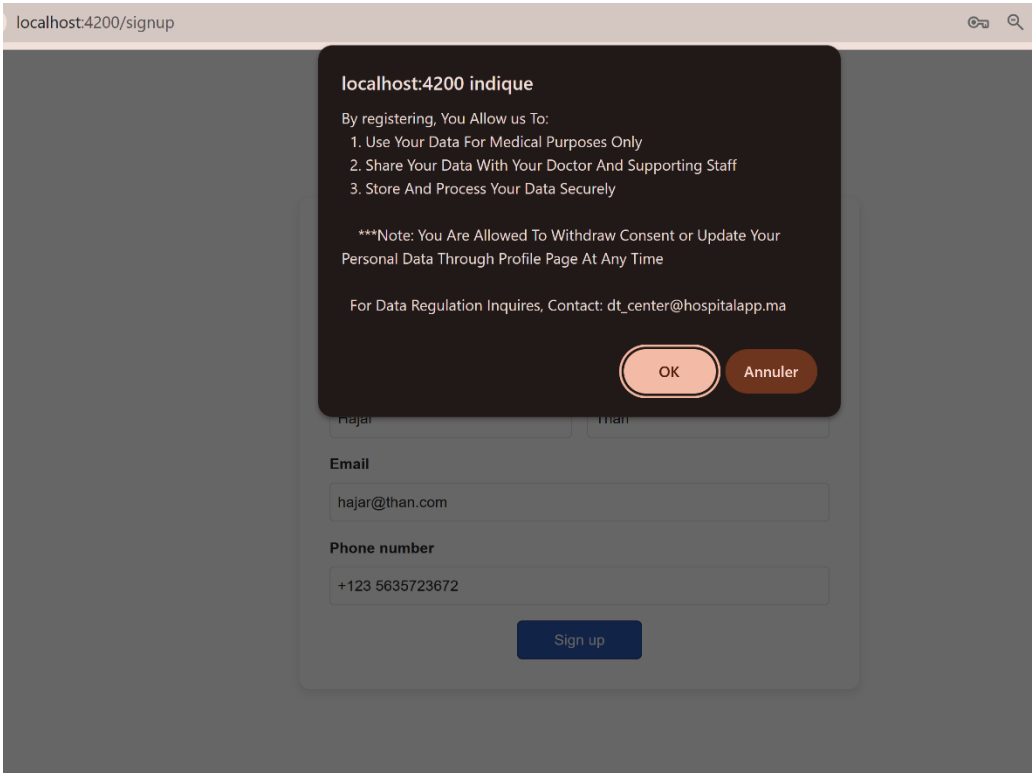
    - Vulnerable dependencies: python-jose@3.4.0
    - Use default credential for admin account due to the lack of changing password function
    - Login function has no limitation for fail attempts
- Frontend:
    - Access to ng serve related files is not restricted
    - Information exposure for static files in frontend

VI. Suggestions For Improvement

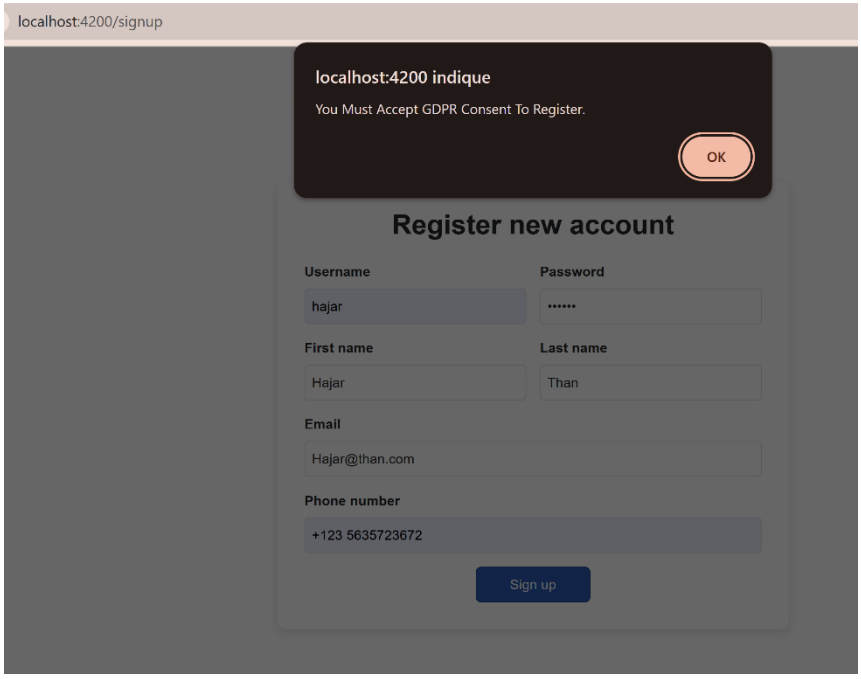
- Entire Database Encryption
- Resolve All Vulnerabilities in The Scanning Report
- Add Logs Auditing
- Rate Limiting Feature
- Implement Refresh Token
- Implement Change Password Functionality

VII. GDPR Regulations

Since in our project we are processing sensitive personal data, we had to think about the General Data Protection Regulation (GDPR), and for that, we have added a consent alert while registering asking for user consent and specifying their rights, our purpose behind data collection, and its processing. The consent is asked in clear language, and users are free to accept it or not.



In case consent is given, the registration is finalized; otherwise, the registration is cancelled.



VIII. AI Usage

- Name: ChatGPT – 4o
  - Purpose:
    - Assist coding and debugging in this project
    - Search information and give suggestions for framework, security practices and vulnerabilities remediation.
- Name: Grammarly
  - Purpose
    - Check and fix our grammar and syntax

We are aware that we are totally responsible for the entire content of the project, including the parts generated by AI, and accept responsibility for any violations of the ethical standards of publications.