Universidade de Aveiro
Mestrado em Cibersegurança
Código: 41783 - Identification, Authentication and Authorization
Responsible: João Paulo Silva Barraca

Flexible, Risk Aware Authentication System

Thursday 23rd May, 2024

Ricardo Covelo (102668) - Telmo Sauce (104428)

Contents

1 Introduction

As the second part of the IAA class project, this report aims to show how we applied the ideas from the last report. It also explains the changes we made based on the professor's feedback and some new problems we discovered.

2 Changes and additions

2.1 MFA

As the professor pointed out, security questions are not a reliable form of authentication and need to be replaced with a better method. This is because if someone can find the answers to these questions, they can access the account. Often, the answers to these questions can be discoverable through OSINT methods.

Therefore, we decided to replace this authentication method with TOTP (Time-Based One-Time Password). This method uses both an identity provider (IDP) and a separate device that shares a secret key. The Device generates a unique code every 30 seconds which is then validated by IDP.

2.2 Flow

While working on the project, we noticed some missing details in the previous version, such as missing columns in the database and missing information in the data flow.

In response we created a new version to correct these mistakes, the overall behavior of the system remains the same.

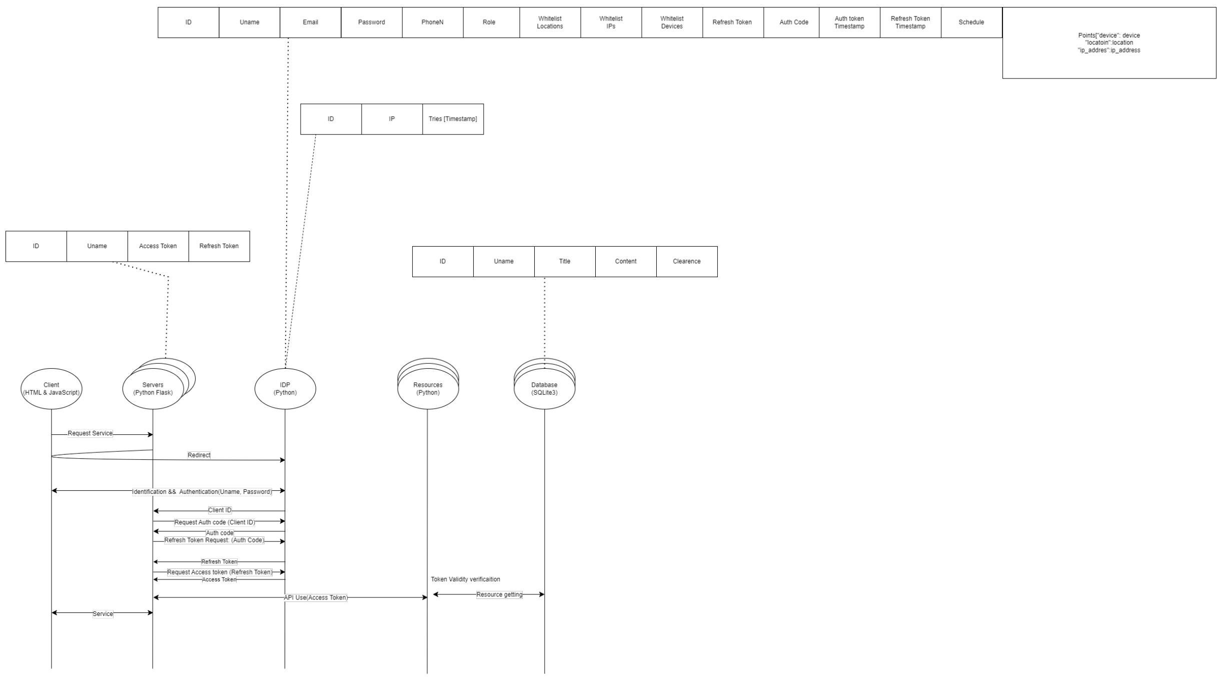


Figure 1: Updated Data Flow

The changes are as follows:

- There is only one IDP database, and the application that registers said user will be inferred by the role of the user.
- A new column with the same name will replace the ”Points” column. This new column will contain a JSON file that represents all sessions of a user account. Risk will be calculated by session instead of by account, sessions have the following format ”{device}:{location}:{ip_address}”. This way, if someone tries to brute force user X’s account, the number of authentication methods user X needs to perform will not change.

The authentication flow will be explained in more detail in the Authentication Flow Section.

2.3 Risk calculation

There also will be changes in how the risk points are calculated. Beyond the previously mentioned change where the points are calculated by session instead of by account, the points will now be changed to 0 for each session that manages to log in and will not decrease with time. The points will be recalculated every time the refresh token expires, and the user will be redirected to the login page in case the IDP needs re-authentication by the user.

2.4 Risk Factors

It was decided not to implement the scheduling risk factor due to its complexity.

In a practical setting, implementing this process typically involves leveraging machine learning algorithms to determine the typical login schedule of a user. Afterward, the system can assign the score to the user sessions that fall outside of their regular usage hours.

3 Implementation

3.1 Debugging

Before explaining and demonstrating the usage of the service, there is a need to clarify some decisions that were made, which although they may not be secure, are present for the sake of debugging or ease of implementation.

- Use of HTTP instead of HTTPS
- Debug field where the IP, device, and Location can be changed.
- Use of symmetric keys in JWT Tokens instead of asymmetric ones.
- Totp uses the username as its secret key.
- Sensitive fields such as passwords are not hashed before being stored in the database.

The real scenario to get the IP is to use ”request.remote_addr”, for the device it would be necessary to get the filed ”User-Agent” and lastly for the the location, it could be calculated based on the IP using an API such as ”https://ipinfo.io/”.

Considering the asymmetric keys, the IDP should have a private key to sign the JWT token, and the resource server should have the corresponding public key to validate the token.

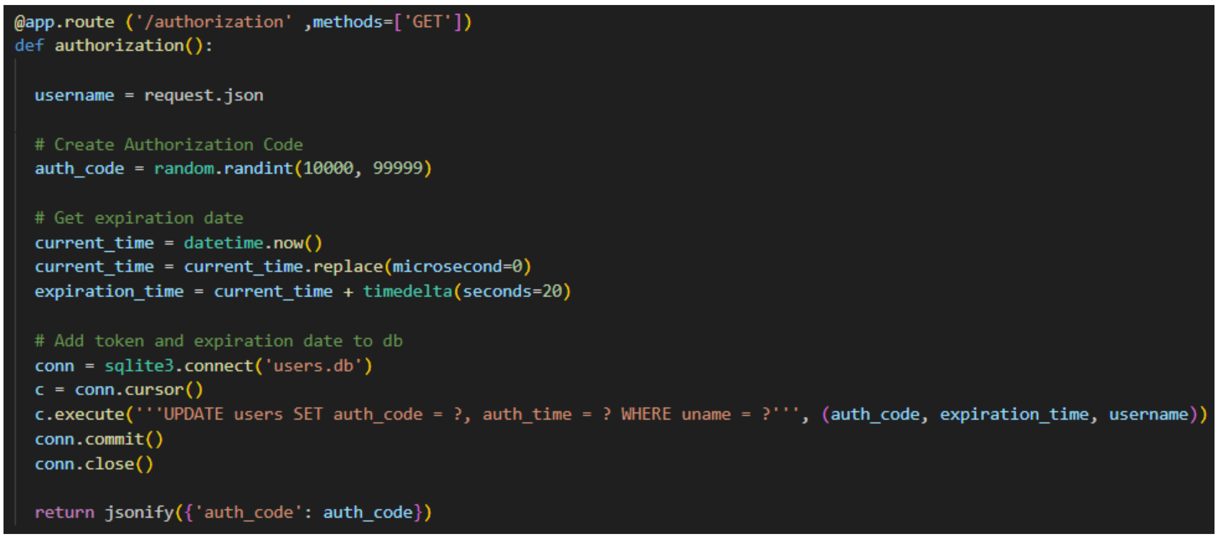
The TOTP secret key is currently insecure because it uses the users name. To enhance security, users should create a stronger password and be able to store it in the identity provider. For an even more secure implementation, the Diffie–Hellman Algorithm can be used to generate the shared key.

3.2 Tokens

The tokens are implemented in two different ways, one uses a random number and the other one is using a JWT token.

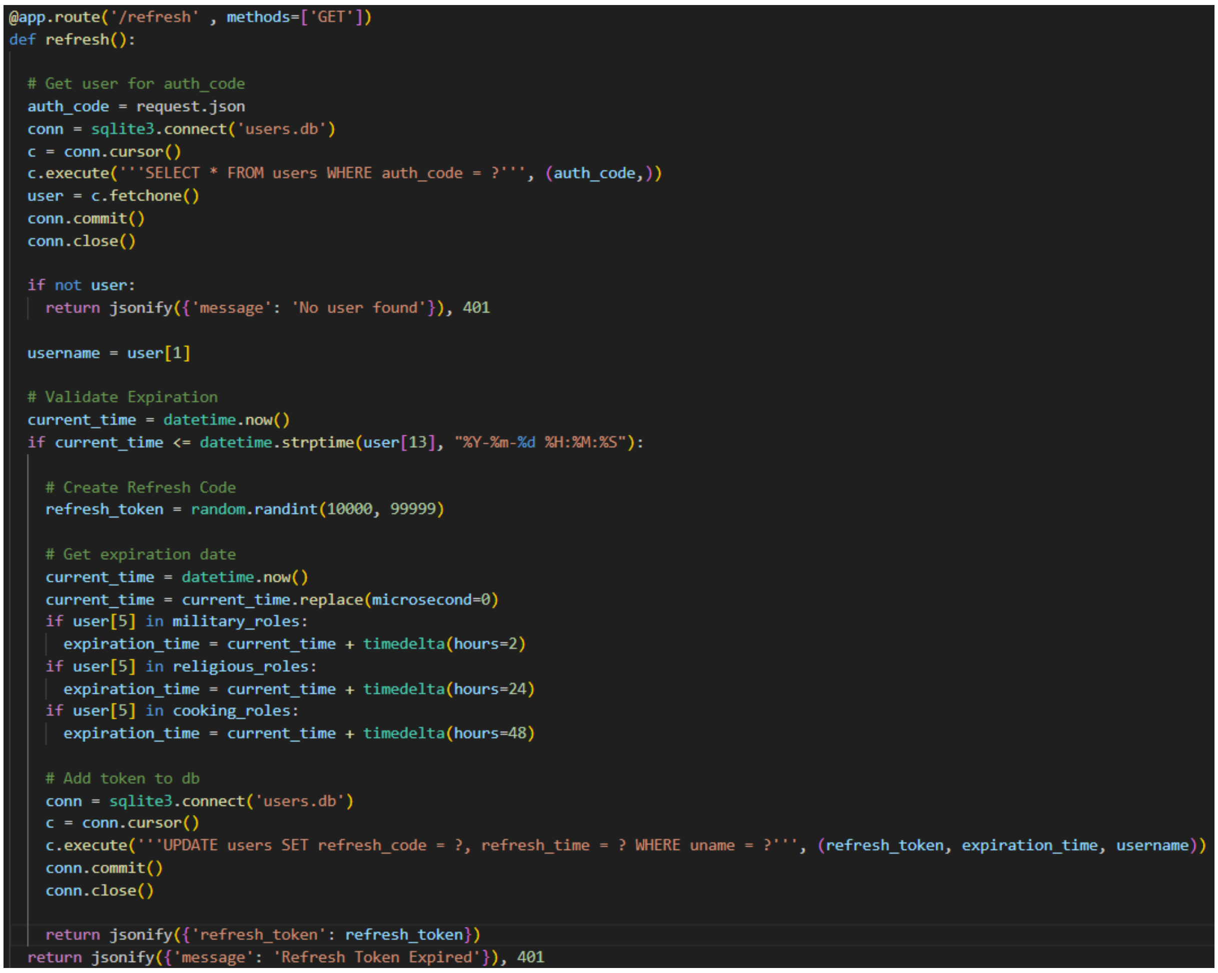
The Auth code and the Refresh token are implemented as random numbers, between 10000 and 99999, that are associated with the user in the IDP.

As these tokens rarely leave the IDP and are verified internally by the IDP, forging them becomes more difficult, so there is no necessity for using a JWT token. Additionally, we assume there is a trusted relationship between the IDP and the application, and that communication occurs over HTTPS.



```python
@app.route ('/authorization' ,methods=['GET'])
def authorization():

    username = request.json

    # Create Authorization Code
    auth_code = random.randint(10000, 99999)

    # Get expiration date
    current_time = datetime.now()
    current_time = current_time.replace(microsecond=0)
    expiration_time = current_time + timedelta(seconds=20)

    # Add token and expiration date to db
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''UPDATE users SET auth_code = ?, auth_time = ? WHERE uname = ?''', (auth_code, expiration_time, username))
    conn.commit()
    conn.close()

    return jsonify({'auth_code': auth_code})
```

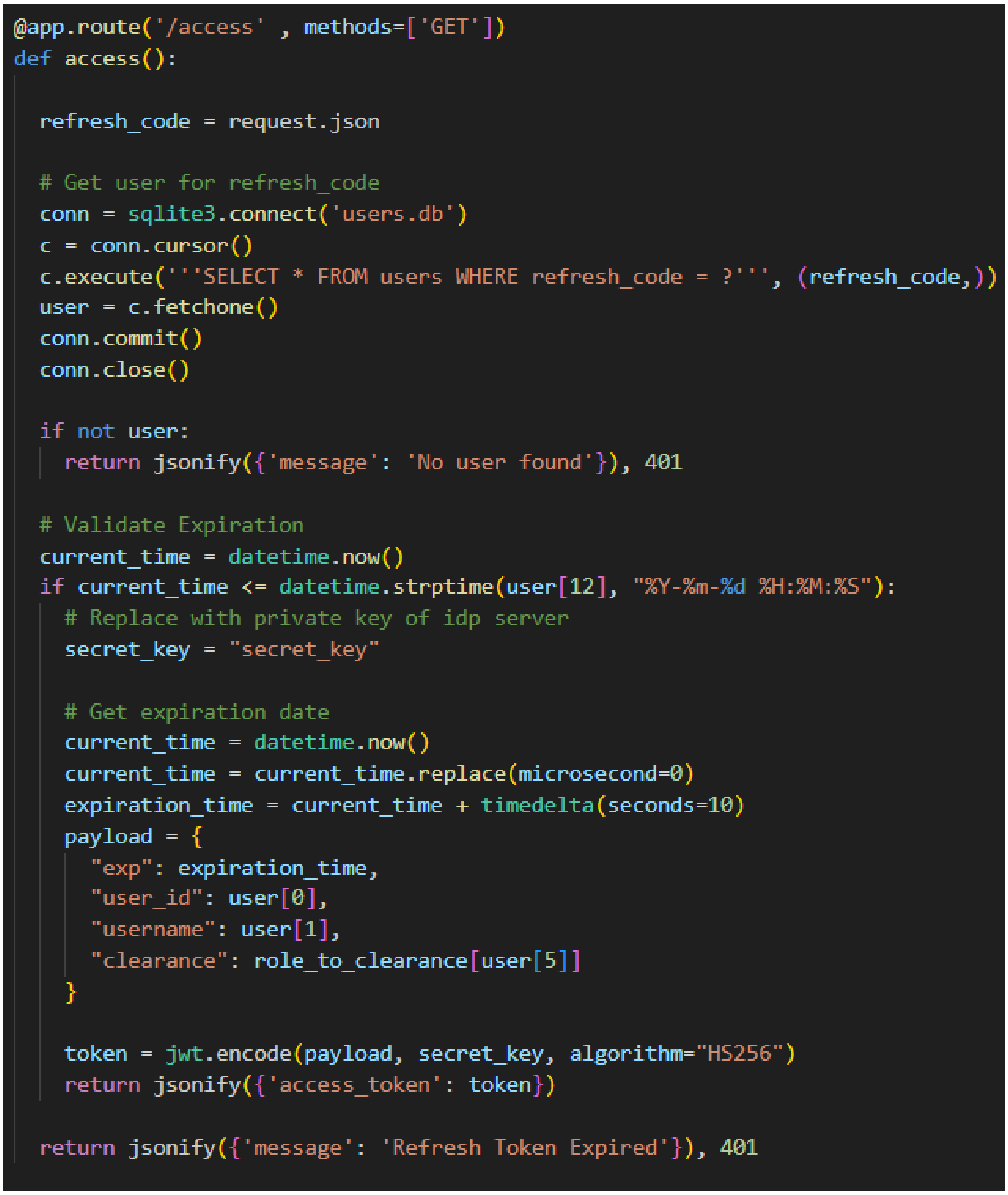
Figure 2: Authorization Code



```python
@app.route('/refresh' , methods=['GET'])
def refresh():

    # Get user for auth_code
    auth_code = request.json
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''SELECT * FROM users WHERE auth_code = ?''', (auth_code,))
    user = c.fetchone()
    conn.commit()
    conn.close()

    if not user:
        return jsonify({'message': 'No user found'}), 401

    username = user[1]

    # Validate Expiration
    current_time = datetime.now()
    if current_time <= datetime.strptime(user[13], "%Y-%m-%d %H:%M:%S"):

        # Create Refresh Code
        refresh_token = random.randint(10000, 99999)

        # Get expiration date
        current_time = datetime.now()
        current_time = current_time.replace(microsecond=0)
        if user[5] in military_roles:
            expiration_time = current_time + timedelta(hours=2)
        if user[5] in religious_roles:
            expiration_time = current_time + timedelta(hours=24)
        if user[5] in cooking_roles:
            expiration_time = current_time + timedelta(hours=48)

        # Add token to db
        conn = sqlite3.connect('users.db')
        c = conn.cursor()
        c.execute('''UPDATE users SET refresh_code = ?, refresh_time = ? WHERE uname = ?''', (refresh_token, expiration_time, username))
        conn.commit()
        conn.close()

        return jsonify({'refresh_token': refresh_token})
    return jsonify({'message': 'Refresh Token Expired'}), 401
```

Figure 3: Refresh Token

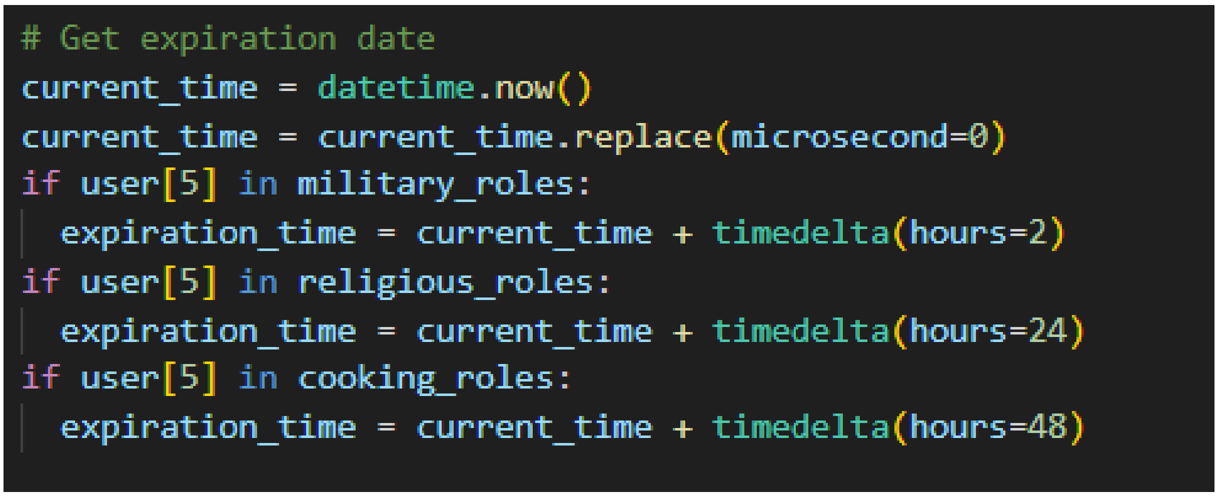
The Access Token is the only one implemented with JTW, since if it was not used, a database would need to be **shared** between the IDP and the resource server, since the resource server would later need to validate the token, using a singed token helps to avoid this problem. Also, the resource server needs information about the user that only the IDP can provide, which can be stored on the JWT.



```
@app.route('/access' , methods=['GET'])
def access():

  refresh_code = request.json

  # Get user for refresh_code
  conn = sqlite3.connect('users.db')
  c = conn.cursor()
  c.execute('''SELECT * FROM users WHERE refresh_code = ?''', (refresh_code,))
  user = c.fetchone()
  conn.commit()
  conn.close()

  if not user:
    return jsonify({'message': 'No user found'}), 401

  # Validate Expiration
  current_time = datetime.now()
  if current_time <= datetime.strptime(user[12], "%Y-%m-%d %H:%M:%S"):
    # Replace with private key of idp server
    secret_key = "secret_key"

    # Get expiration date
    current_time = datetime.now()
    current_time = current_time.replace(microsecond=0)
    expiration_time = current_time + timedelta(seconds=10)
    payload = {
      "exp": expiration_time,
      "user_id": user[0],
      "username": user[1],
      "clearance": role_to_clearance[user[5]]
    }

    token = jwt.encode(payload, secret_key, algorithm="HS256")
    return jsonify({'access_token': token})

  return jsonify({'message': 'Refresh Token Expired'}), 401
```

Figure 4: Access Token

Each application has its own refresh token expiration. For the **less secure** application, the refresh token expires after 48 hours. For the more secure applications, the expiration periods are shorter, with the **secure** application setting the expiration to 24 hours, and the **very secure** one reducing it to just 2 hours.



```python
# Get expiration date
current_time = datetime.now()
current_time = current_time.replace(microsecond=0)
if user[5] in military_roles:
    expiration_time = current_time + timedelta(hours=2)
if user[5] in religious_roles:
    expiration_time = current_time + timedelta(hours=24)
if user[5] in cooking_roles:
    expiration_time = current_time + timedelta(hours=48)
```

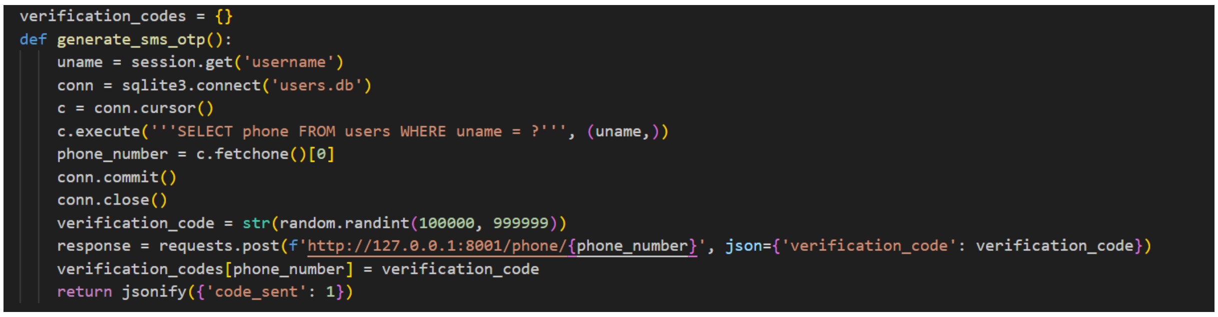
Figure 5: Refresh token expiration

3.3 MFA

As most of our alternative methods of authentication require a second device for easier implementation, we created a webpage to simulate the usage of these devices. Additionally, the secret code is displayed on the terminal as well.

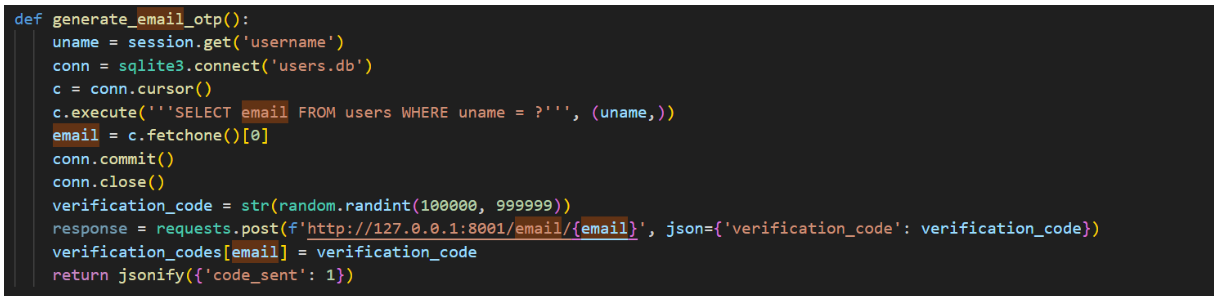
3.3.1 Phone and Email OTP

As said before, we created a webpage to simulate the usage of a phone. This OTP is based on the IDP sending to the ”Phone” a **random number** with 6 digits, that are received and displayed. Then the IDP asks for said code as confirmation for the codes being received.



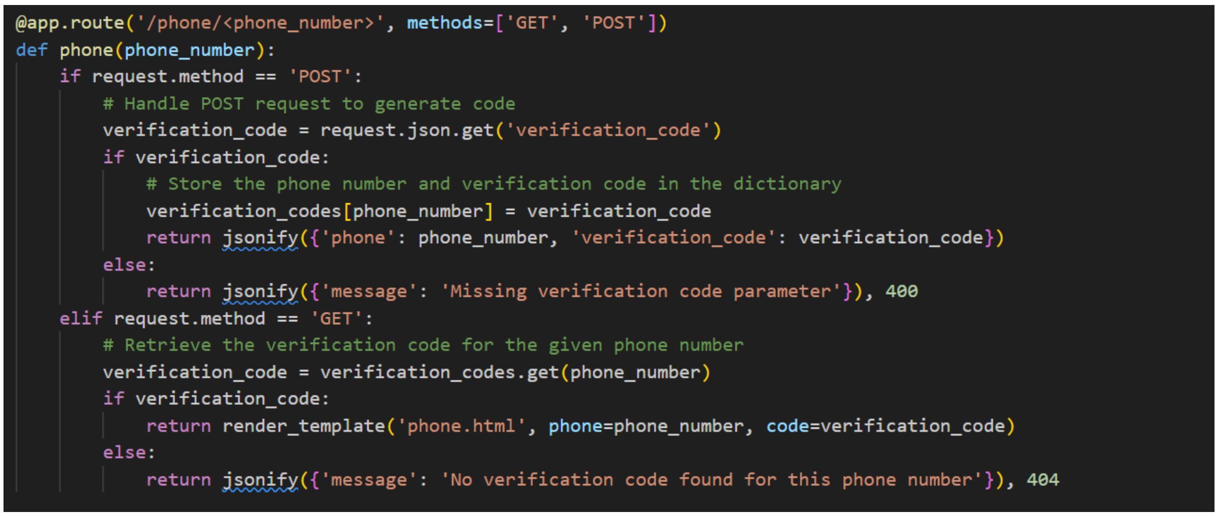
```python
verification_codes = {}
def generate_sms_otp():
    uname = session.get('username')
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''SELECT phone FROM users WHERE uname = ?''', (uname,))
    phone_number = c.fetchone()[0]
    conn.commit()
    conn.close()
    verification_code = str(random.randint(100000, 999999))
    response = requests.post(f'http://127.0.0.1:8001/phone/{phone_number}', json={'verification_code': verification_code})
    verification_codes[phone_number] = verification_code
    return jsonify({'code_sent': 1})
```

Figure 6: IDP Phone OTP



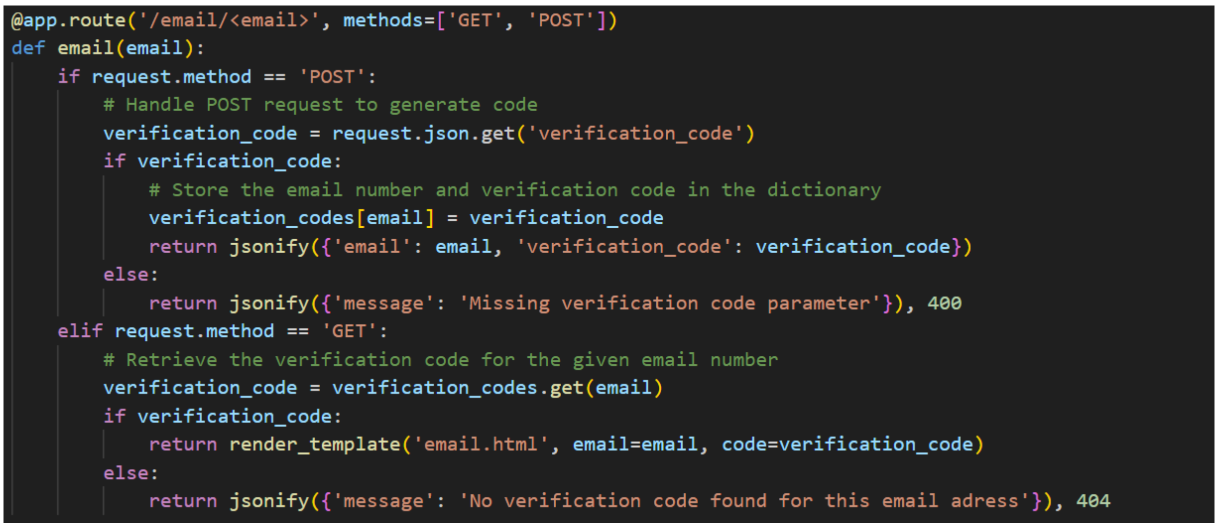
```python
def generate_email_otp():
    uname = session.get('username')
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''SELECT email FROM users WHERE uname = ?''', (uname,))
    email = c.fetchone()[0]
    conn.commit()
    conn.close()
    verification_code = str(random.randint(100000, 999999))
    response = requests.post(f'http://127.0.0.1:8001/email/{email}', json={'verification_code': verification_code})
    verification_codes[email] = verification_code
    return jsonify({'code_sent': 1})
```

Figure 7: IDP Email OTP



```python
@app.route('/phone/<phone_number>', methods=['GET', 'POST'])
def phone(phone_number):
    if request.method == 'POST':
        # Handle POST request to generate code
        verification_code = request.json.get('verification_code')
        if verification_code:
            # Store the phone number and verification code in the dictionary
            verification_codes[phone_number] = verification_code
            return jsonify({'phone': phone_number, 'verification_code': verification_code})
        else:
            return jsonify({'message': 'Missing verification code parameter'}), 400
    elif request.method == 'GET':
        # Retrieve the verification code for the given phone number
        verification_code = verification_codes.get(phone_number)
        if verification_code:
            return render_template('phone.html', phone=phone_number, code=verification_code)
        else:
            return jsonify({'message': 'No verification code found for this phone number'}), 404
```

Figure 8: MFA Phone OTP

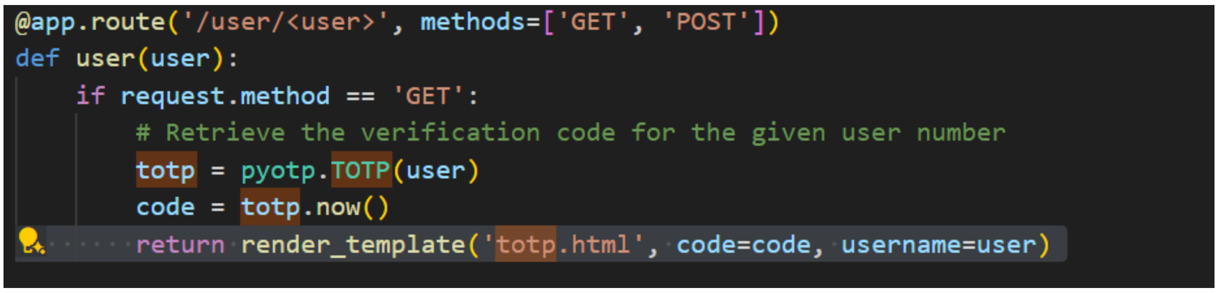


```python
@app.route('/email/<email>', methods=['GET', 'POST'])
def email(email):
    if request.method == 'POST':
        # Handle POST request to generate code
        verification_code = request.json.get('verification_code')
        if verification_code:
            # Store the email number and verification code in the dictionary
            verification_codes[email] = verification_code
            return jsonify({'email': email, 'verification_code': verification_code})
        else:
            return jsonify({'message': 'Missing verification code parameter'}), 400
    elif request.method == 'GET':
        # Retrieve the verification code for the given email number
        verification_code = verification_codes.get(email)
        if verification_code:
            return render_template('email.html', email=email, code=verification_code)
        else:
            return jsonify({'message': 'No verification code found for this email adress'}), 404
```

Figure 9: MFA Email OTP

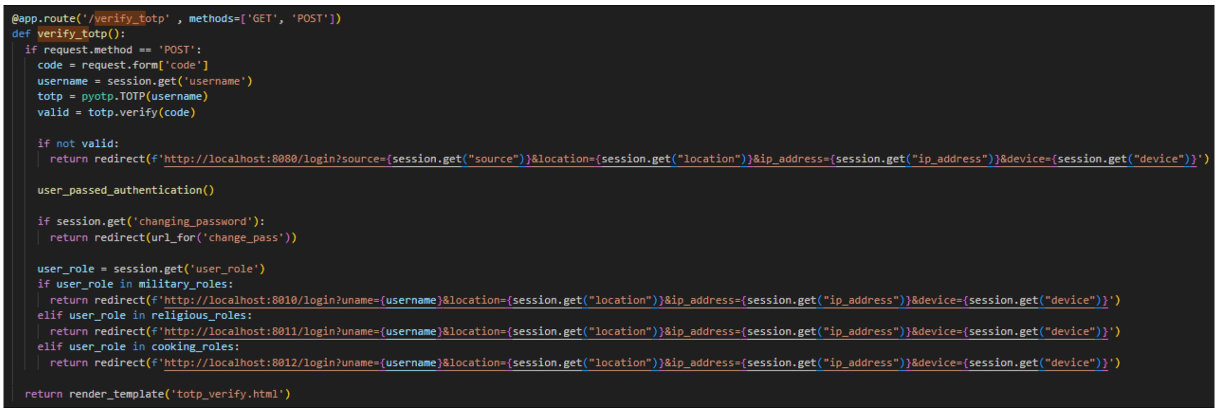
3.3.2 TOPT

TOPT is implemented by generating a code on a Device using the owner's name as its secret key, It would usually be used as a nonce but for the sake of simplicity of implementation, the name was used instead. Then the IDP will verify the Token using the corresponding key, the user only has a window of 30 seconds to insert the code before it is considered invalid again.



```python
@app.route('/user/<user>', methods=['GET', 'POST'])
def user(user):
    if request.method == 'GET':
        # Retrieve the verification code for the given user number
        totp = pyotp.TOTP(user)
        code = totp.now()
        return render_template('totp.html', code=code, username=user)
```

Figure 10: TOPT Creation



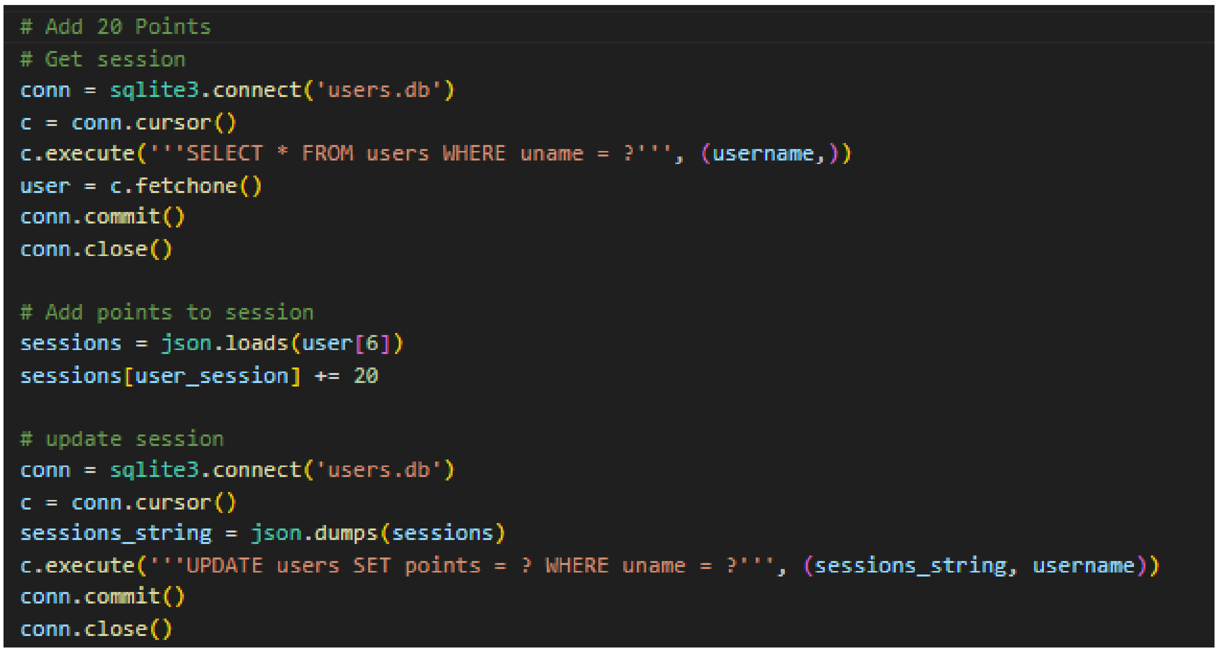
```python
@app.route('/verify_totp' , methods=['GET', 'POST'])
def verify_totp():
  if request.method == 'POST':
    code = request.form['code']
    username = session.get('username')
    totp = pyotp.TOTP(username)
    valid = totp.verify(code)

    if not valid:
      return redirect(f'http://localhost:8080/login?source={session.get("source")}&location={session.get("location")}&ip_address={session.get("ip_address")}&device={session.get("device")}')

    user_passed_authentication()

    if session.get('changing_password'):
      return redirect(url_for('change_pass'))

    user_role = session.get('user_role')
    if user_role in military_roles:
      return redirect(f'http://localhost:8010/login?uname={username}&location={session.get("location")}&ip_address={session.get("ip_address")}&device={session.get("device")}')
    elif user_role in religious_roles:
      return redirect(f'http://localhost:8011/login?uname={username}&location={session.get("location")}&ip_address={session.get("ip_address")}&device={session.get("device")}')
    elif user_role in cooking_roles:
      return redirect(f'http://localhost:8012/login?uname={username}&location={session.get("location")}&ip_address={session.get("ip_address")}&device={session.get("device")}')

  return render_template('totp_verify.html')
```

Figure 11: TOPT Verify

3.4 Risk factors

3.4.1 Brute Force

Brute Force attempts to trigger the following Snippet of code. Where it is added 20 points to the overall score of the session for every wrong attempt.



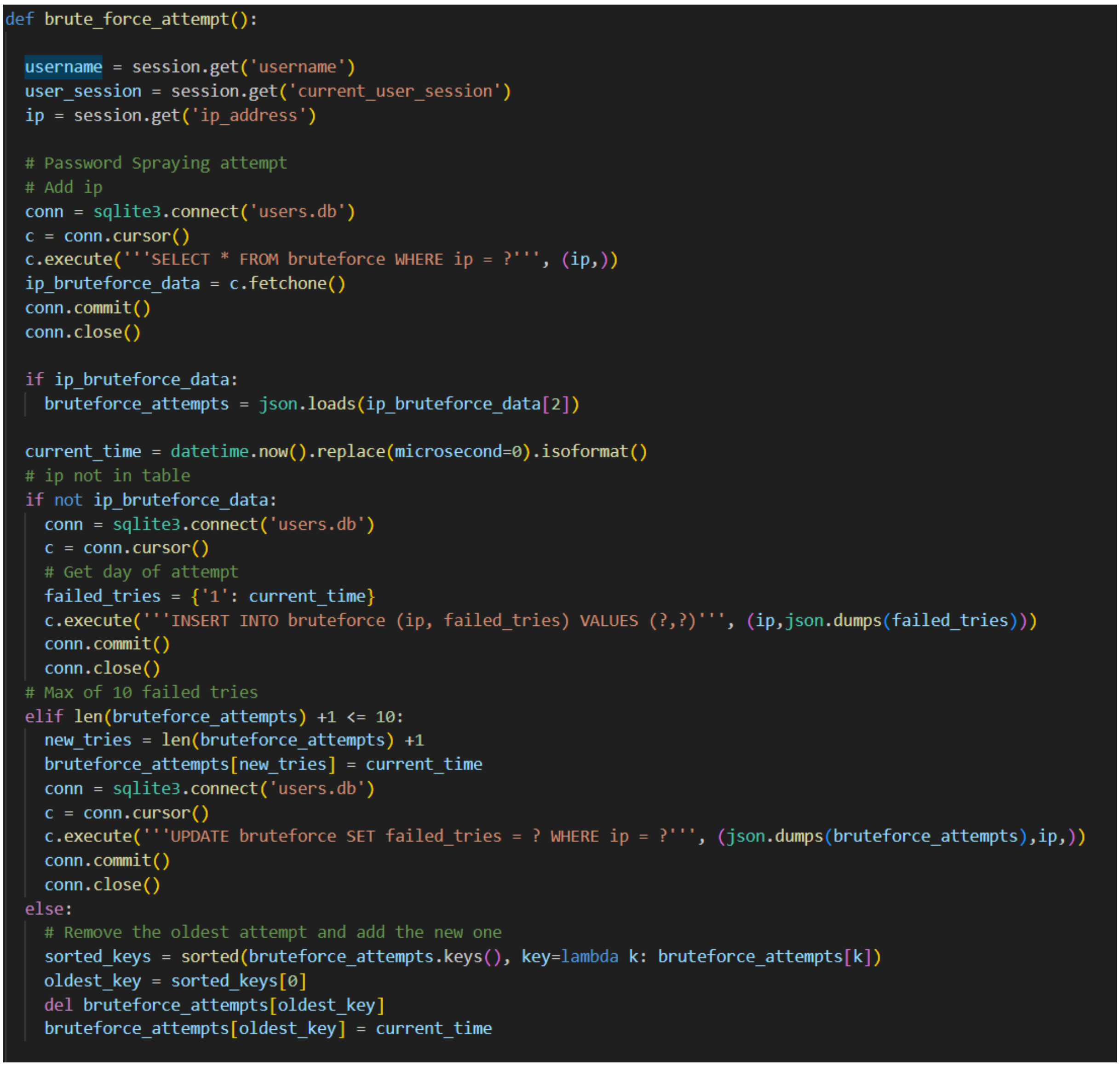
```
# Add 20 Points
# Get session
conn = sqlite3.connect('users.db')
c = conn.cursor()
c.execute('''SELECT * FROM users WHERE uname = ?''', (username,))
user = c.fetchone()
conn.commit()
conn.close()

# Add points to session
sessions = json.loads(user[6])
sessions[user_session] += 20

# update session
conn = sqlite3.connect('users.db')
c = conn.cursor()
sessions_string = json.dumps(sessions)
c.execute('''UPDATE users SET points = ? WHERE uname = ?''', (sessions_string, username))
conn.commit()
conn.close()
```

Figure 12: Brute Force Attempt

3.4.2 Password Spraying

Password spraying attempts are stored in a different table and so, points associated with them are calculated further down the code and added to the final score. As stated before, only the last 10 different tries are stored in the table and will be deleted after two weeks.

Password spraying attempts are stored in a different table and so, as stated on the first assignment the last 10 different tries are stored in the table. If an 11th attempt occurs the older attempt stored is removed and inserted the new one.

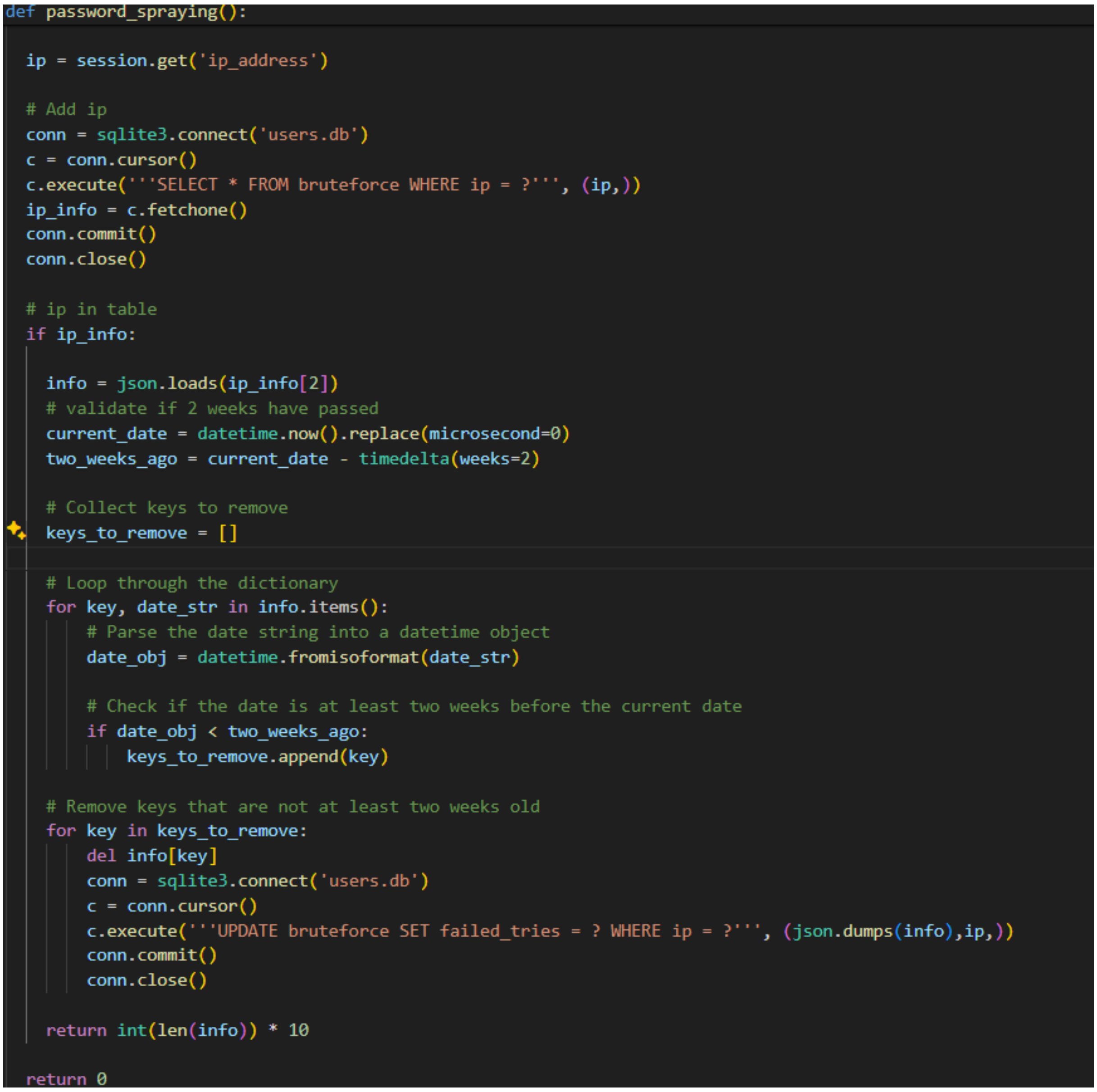


```python
def brute_force_attempt():

  username = session.get('username')
  user_session = session.get('current_user_session')
  ip = session.get('ip_address')

  # Password Spraying attempt
  # Add ip
  conn = sqlite3.connect('users.db')
  c = conn.cursor()
  c.execute('''SELECT * FROM bruteforce WHERE ip = ?''', (ip,))
  ip_bruteforce_data = c.fetchone()
  conn.commit()
  conn.close()

  if ip_bruteforce_data:
    bruteforce_attempts = json.loads(ip_bruteforce_data[2])

  current_time = datetime.now().replace(microsecond=0).isoformat()
  # ip not in table
  if not ip_bruteforce_data:
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    # Get day of attempt
    failed_tries = {'1': current_time}
    c.execute('''INSERT INTO bruteforce (ip, failed_tries) VALUES (?,?)''', (ip,json.dumps(failed_tries)))
    conn.commit()
    conn.close()
  # Max of 10 failed tries
  elif len(bruteforce_attempts) +1 <= 10:
    new_tries = len(bruteforce_attempts) +1
    bruteforce_attempts[new_tries] = current_time
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''UPDATE bruteforce SET failed_tries = ? WHERE ip = ?''', (json.dumps(bruteforce_attempts),ip,))
    conn.commit()
    conn.close()
  else:
    # Remove the oldest attempt and add the new one
    sorted_keys = sorted(bruteforce_attempts.keys(), key=lambda k: bruteforce_attempts[k])
    oldest_key = sorted_keys[0]
    del bruteforce_attempts[oldest_key]
    bruteforce_attempts[oldest_key] = current_time
```

Figure 13: Password Spraying points Calculation

Every time the IDP needs to calculate the points of a certain session It will trigger the following function, it is also hear that the IDP removes attempts that have passed 2 weeks.

Each time the IDP needs to compute the points for a specific session, it will trigger the following function. Additionally, the IDP removes the attempts that occurred more than two weeks ago.

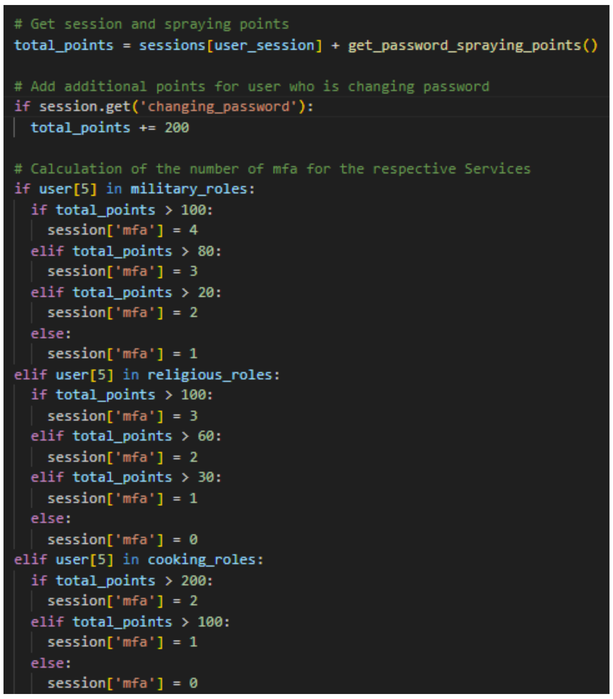


```
def password_spraying():

    ip = session.get('ip_address')

    # Add ip
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''SELECT * FROM bruteforce WHERE ip = ?''', (ip,))
    ip_info = c.fetchone()
    conn.commit()
    conn.close()

    # ip in table
    if ip_info:

        info = json.loads(ip_info[2])
        # validate if 2 weeks have passed
        current_date = datetime.now().replace(microsecond=0)
        two_weeks_ago = current_date - timedelta(weeks=2)

        # Collect keys to remove
        keys_to_remove = []

        # Loop through the dictionary
        for key, date_str in info.items():
            # Parse the date string into a datetime object
            date_obj = datetime.fromisoformat(date_str)

            # Check if the date is at least two weeks before the current date
            if date_obj < two_weeks_ago:
                keys_to_remove.append(key)

        # Remove keys that are not at least two weeks old
        for key in keys_to_remove:
            del info[key]
            conn = sqlite3.connect('users.db')
            c = conn.cursor()
            c.execute('''UPDATE bruteforce SET failed_tries = ? WHERE ip = ?''', (json.dumps(info),ip,))
            conn.commit()
            conn.close()

        return int(len(info)) * 10

    return 0
```

Figure 14: Getting Password spraying points

3.4.3 Setting Tempering

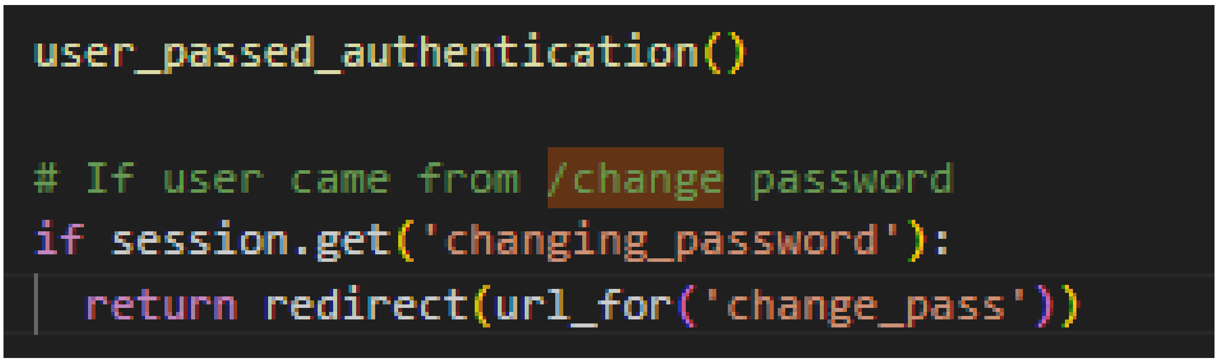
Users on the IDP have the possibility of changing their password. To do this, they are first redirected to the login page to re-authenticate. This action adds 200 points to their overall score for tempering actions, then the session points are recalculated and the number of MFA methods is determined.



```python
# Get session and spraying points
total_points = sessions[user_session] + get_password_spraying_points()

# Add additional points for user who is changing password
if session.get('changing_password'):
    total_points += 200

# Calculation of the number of mfa for the respective Services
if user[5] in military_roles:
    if total_points > 100:
        session['mfa'] = 4
    elif total_points > 80:
        session['mfa'] = 3
    elif total_points > 20:
        session['mfa'] = 2
    else:
        session['mfa'] = 1
elif user[5] in religious_roles:
    if total_points > 100:
        session['mfa'] = 3
    elif total_points > 60:
        session['mfa'] = 2
    elif total_points > 30:
        session['mfa'] = 1
    else:
        session['mfa'] = 0
elif user[5] in cooking_roles:
    if total_points > 200:
        session['mfa'] = 2
    elif total_points > 100:
        session['mfa'] = 1
    else:
        session['mfa'] = 0
```

Figure 15: Points added for change of password

If the user successfully passes all authentication methods, they are redirected to the password change webpage and issued new tokens.

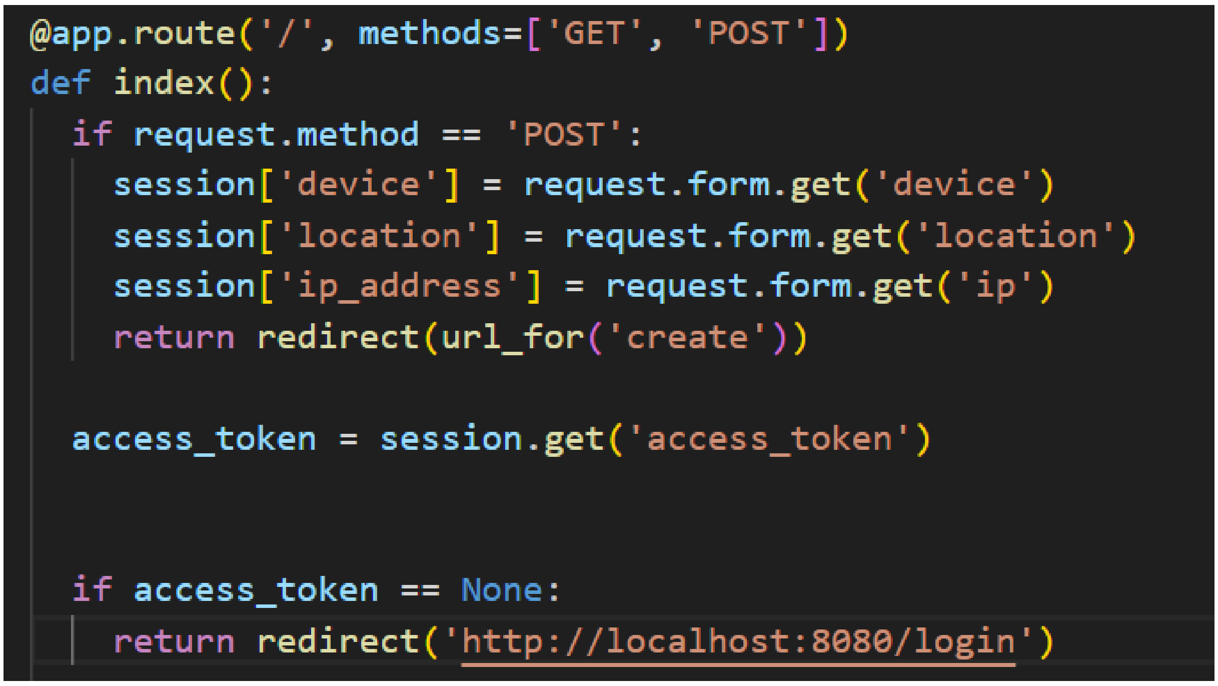


```python
user_passed_authentication()

# If user came from /change password
if session.get('changing_password'):
    return redirect(url_for('change_pass'))
```

Figure 16: Redirect user to change password

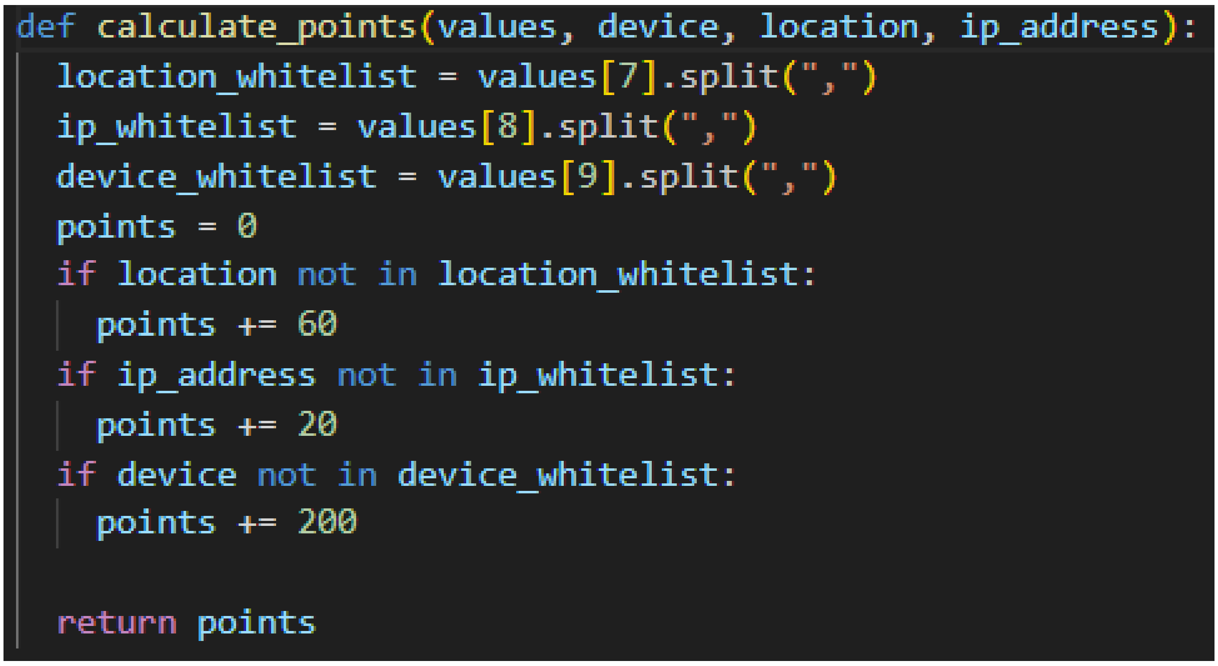
3.5 Authentication Flow

The authentication starts with the app verifying if the user is already authenticated, if not redirects to the IDP where it will authenticate the user.



```python
@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        session['device'] = request.form.get('device')
        session['location'] = request.form.get('location')
        session['ip_address'] = request.form.get('ip')
        return redirect(url_for('create'))

    access_token = session.get('access_token')


    if access_token == None:
        return redirect('http://localhost:8080/login')
```

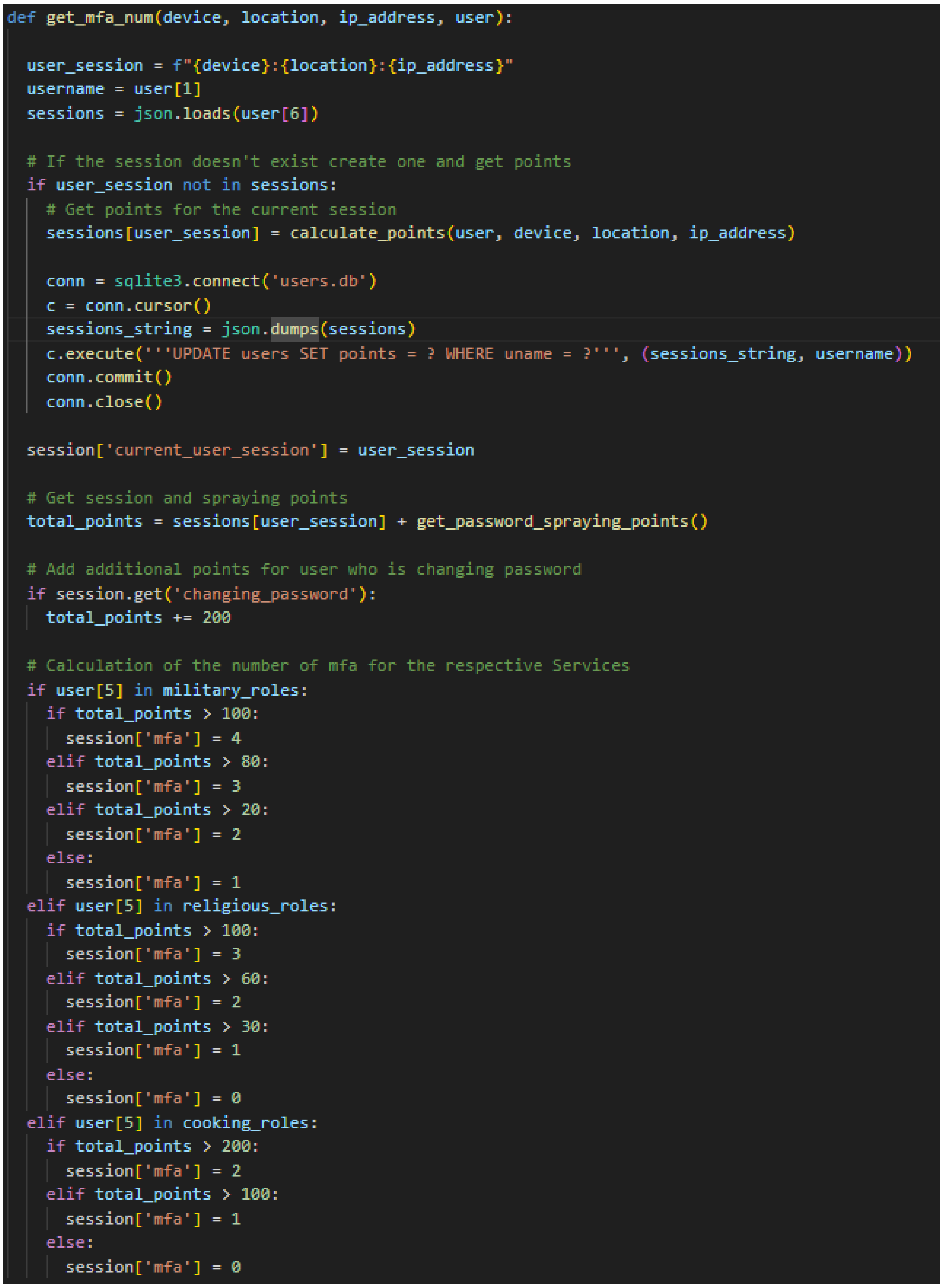
After the user needs to login with his credentials, the IDP will first calculate the user score using the following function, here the IDP validates if the location, device, and IP of the user are on its whitelist if not they add the respective amount of points to the session.



```python
def calculate_points(values, device, location, ip_address):
    location_whitelist = values[7].split(",")
    ip_whitelist = values[8].split(",")
    device_whitelist = values[9].split(",")
    points = 0
    if location not in location_whitelist:
        points += 60
    if ip_address not in ip_whitelist:
        points += 20
    if device not in device_whitelist:
        points += 200

    return points
```

Figure 17: Calculate Points

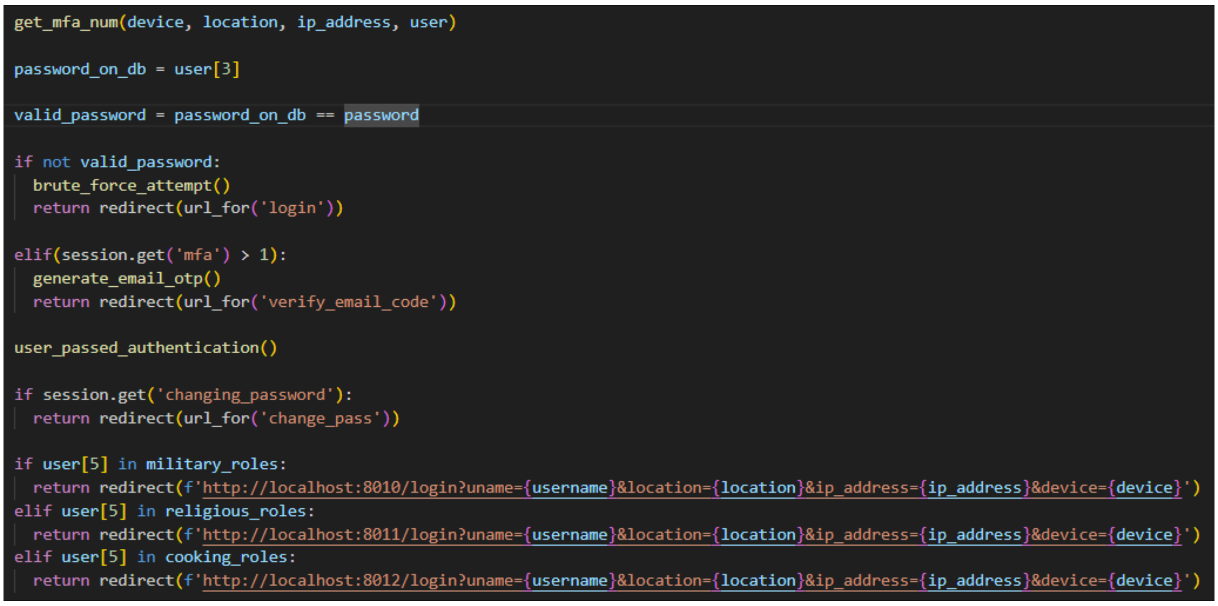
After the points for that session are calculated the IDP will retrieve the points associated with that IP in case it suffered any Password Spraying attack, finally depending on the user's role the IDP determines how many authentication methods are needed for that user to authenticate.



```python
def get_mfa_num(device, location, ip_address, user):

  user_session = f"{device}:{location}:{ip_address}"
  username = user[1]
  sessions = json.loads(user[6])

  # If the session doesn't exist create one and get points
  if user_session not in sessions:
    # Get points for the current session
    sessions[user_session] = calculate_points(user, device, location, ip_address)

    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    sessions_string = json.dumps(sessions)
    c.execute('''UPDATE users SET points = ? WHERE uname = ?''', (sessions_string, username))
    conn.commit()
    conn.close()

  session['current_user_session'] = user_session

  # Get session and spraying points
  total_points = sessions[user_session] + get_password_spraying_points()

  # Add additional points for user who is changing password
  if session.get('changing_password'):
    total_points += 200

  # Calculation of the number of mfa for the respective Services
  if user[5] in military_roles:
    if total_points > 100:
      session['mfa'] = 4
    elif total_points > 80:
      session['mfa'] = 3
    elif total_points > 20:
      session['mfa'] = 2
    else:
      session['mfa'] = 1
  elif user[5] in religious_roles:
    if total_points > 100:
      session['mfa'] = 3
    elif total_points > 60:
      session['mfa'] = 2
    elif total_points > 30:
      session['mfa'] = 1
    else:
      session['mfa'] = 0
  elif user[5] in cooking_roles:
    if total_points > 200:
      session['mfa'] = 2
    elif total_points > 100:
      session['mfa'] = 1
    else:
      session['mfa'] = 0
```

Figure 18: Get number of MFA needed

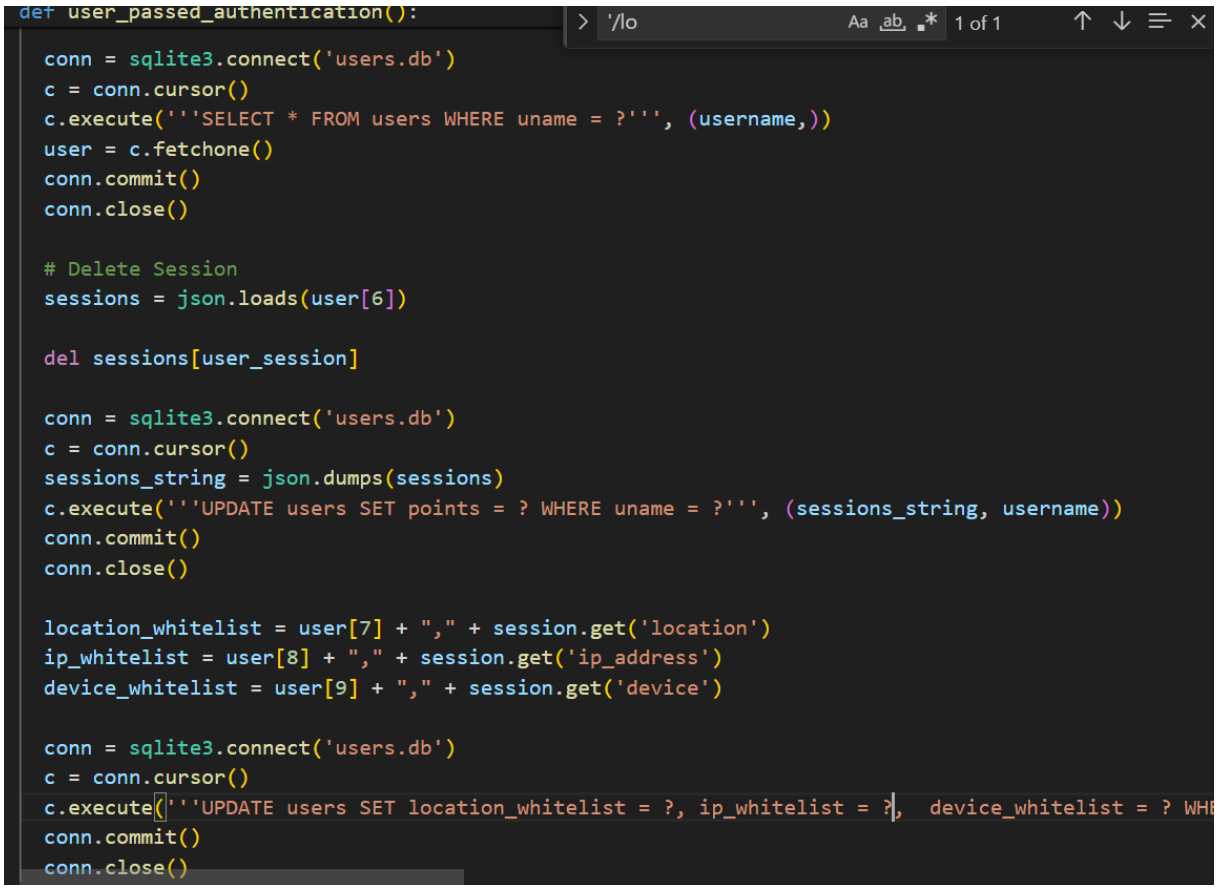
The next step is to validate if the users credentials inserted are valid, if not they trigger the ”brute_force_attempt”. In case the credentials are valid the user is either redirected to the first mfa method which is the email or passes to the final stage which is the ”user_passed_authentication”.



```
get_mfa_num(device, location, ip_address, user)

password_on_db = user[3]

valid_password = password_on_db == password

if not valid_password:
  brute_force_attempt()
  return redirect(url_for('login'))

elif(session.get('mfa') > 1):
  generate_email_otp()
  return redirect(url_for('verify_email_code'))

user_passed_authentication()

if session.get('changing_password'):
  return redirect(url_for('change_pass'))

if user[5] in military_roles:
  return redirect(f'http://localhost:8010/login?uname={username}&location={location}&ip_address={ip_address}&device={device}')
elif user[5] in religious_roles:
  return redirect(f'http://localhost:8011/login?uname={username}&location={location}&ip_address={ip_address}&device={device}')
elif user[5] in cooking_roles:
  return redirect(f'http://localhost:8012/login?uname={username}&location={location}&ip_address={ip_address}&device={device}')
```

Figure 19: Getting the number of Authentication methods and verifying user credentials

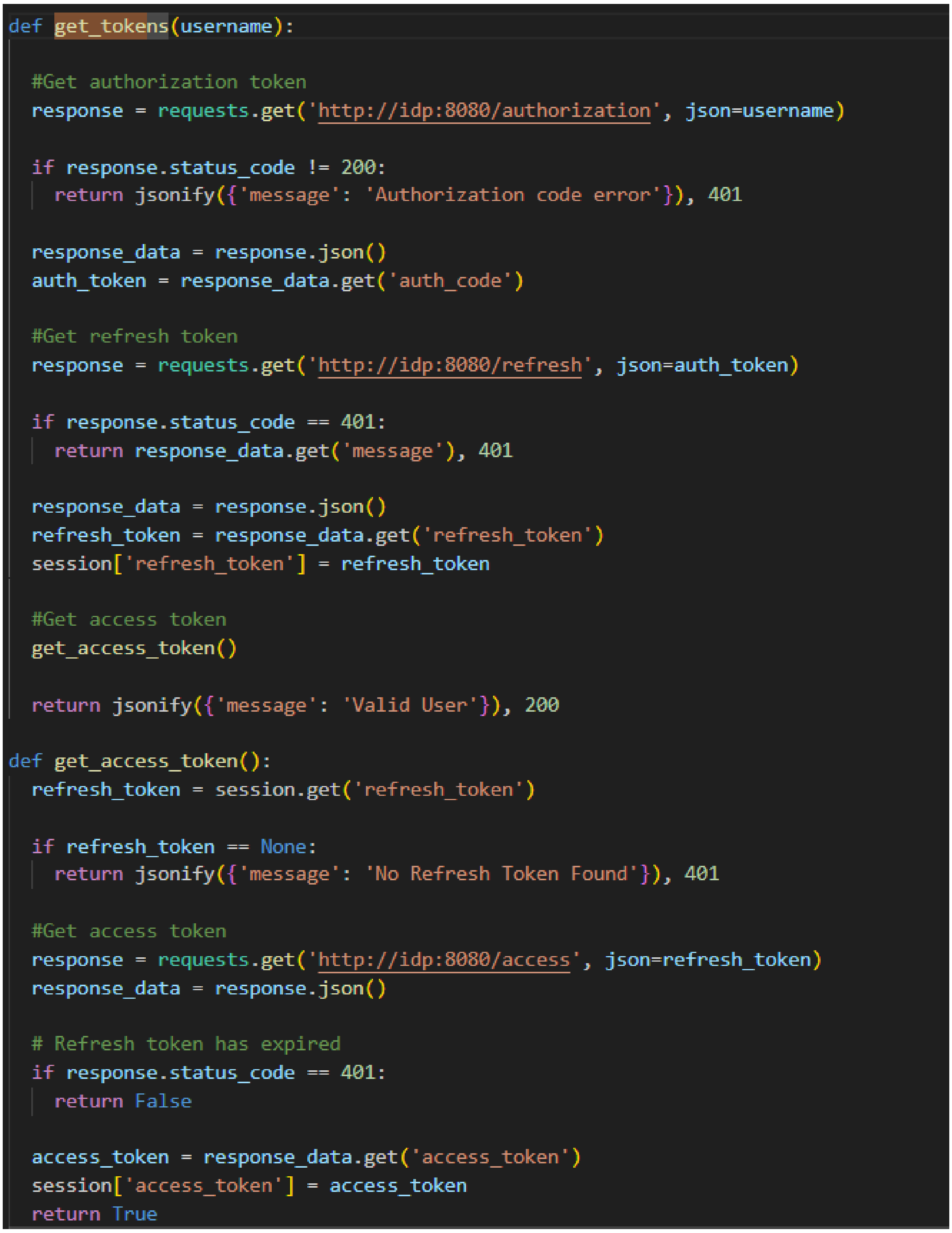
If the user passes all methods necessary, the session is deleted and the device, location, and IP of the user are added to the whitelist. Afterward, the user is redirected to the appropriate service based on their role. The user’s role helps the application determine what route the user should be redirected to.



```
def user_passed_authentication():
    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''SELECT * FROM users WHERE uname = ?''', (username,))
    user = c.fetchone()
    conn.commit()
    conn.close()

    # Delete Session
    sessions = json.loads(user[6])

    del sessions[user_session]

    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    sessions_string = json.dumps(sessions)
    c.execute('''UPDATE users SET points = ? WHERE uname = ?''', (sessions_string, username))
    conn.commit()
    conn.close()

    location_whitelist = user[7] + "," + session.get('location')
    ip_whitelist = user[8] + "," + session.get('ip_address')
    device_whitelist = user[9] + "," + session.get('device')

    conn = sqlite3.connect('users.db')
    c = conn.cursor()
    c.execute('''UPDATE users SET location_whitelist = ?, ip_whitelist = ?,  device_whitelist = ? WHE
    conn.commit()
    conn.close()
```

Figure 20: White listing

Finally, the app requests the IDP for the users tokens. Starting from the authorization which is used to get the refresh token, storing both on the database, with their respective expiration dates. Lastly, the access token is requested using the refresh token.



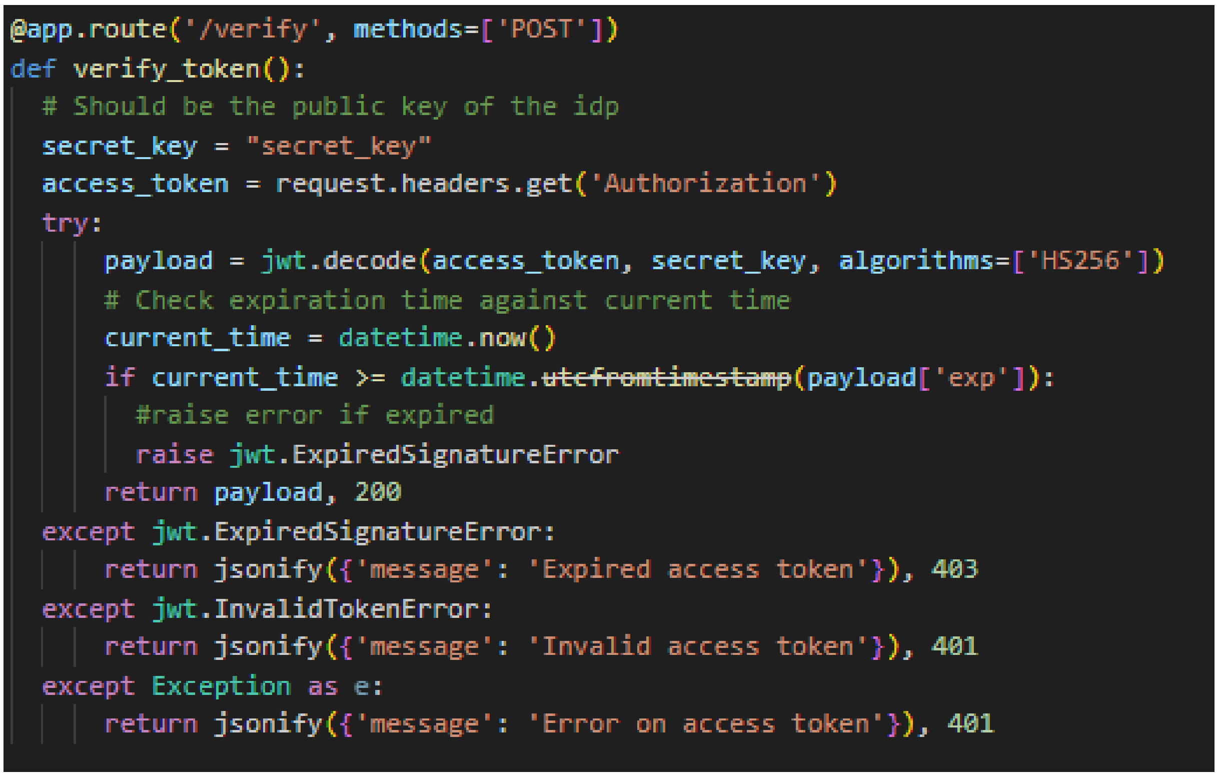
```python
def get_tokens(username):

    #Get authorization token
    response = requests.get('http://idp:8080/authorization', json=username)

    if response.status_code != 200:
        return jsonify({'message': 'Authorization code error'}), 401

    response_data = response.json()
    auth_token = response_data.get('auth_code')

    #Get refresh token
    response = requests.get('http://idp:8080/refresh', json=auth_token)

    if response.status_code == 401:
        return response_data.get('message'), 401

    response_data = response.json()
    refresh_token = response_data.get('refresh_token')
    session['refresh_token'] = refresh_token

    #Get access token
    get_access_token()

    return jsonify({'message': 'Valid User'}), 200

def get_access_token():
    refresh_token = session.get('refresh_token')

    if refresh_token == None:
        return jsonify({'message': 'No Refresh Token Found'}), 401

    #Get access token
    response = requests.get('http://idp:8080/access', json=refresh_token)
    response_data = response.json()

    # Refresh token has expired
    if response.status_code == 401:
        return False

    access_token = response_data.get('access_token')
    session['access_token'] = access_token
    return True
```

Figure 21: App side token request

Considerations as to when should the system recalculate the points were made and we concluded that. Calculating points every time we made a request, in the most secure application we would never leave the login page since 0 points already are equivalent to needing a password and email, so to combat that we decided to only calculate when giving/updating the refresh token.

3.5.1 Token Expiration

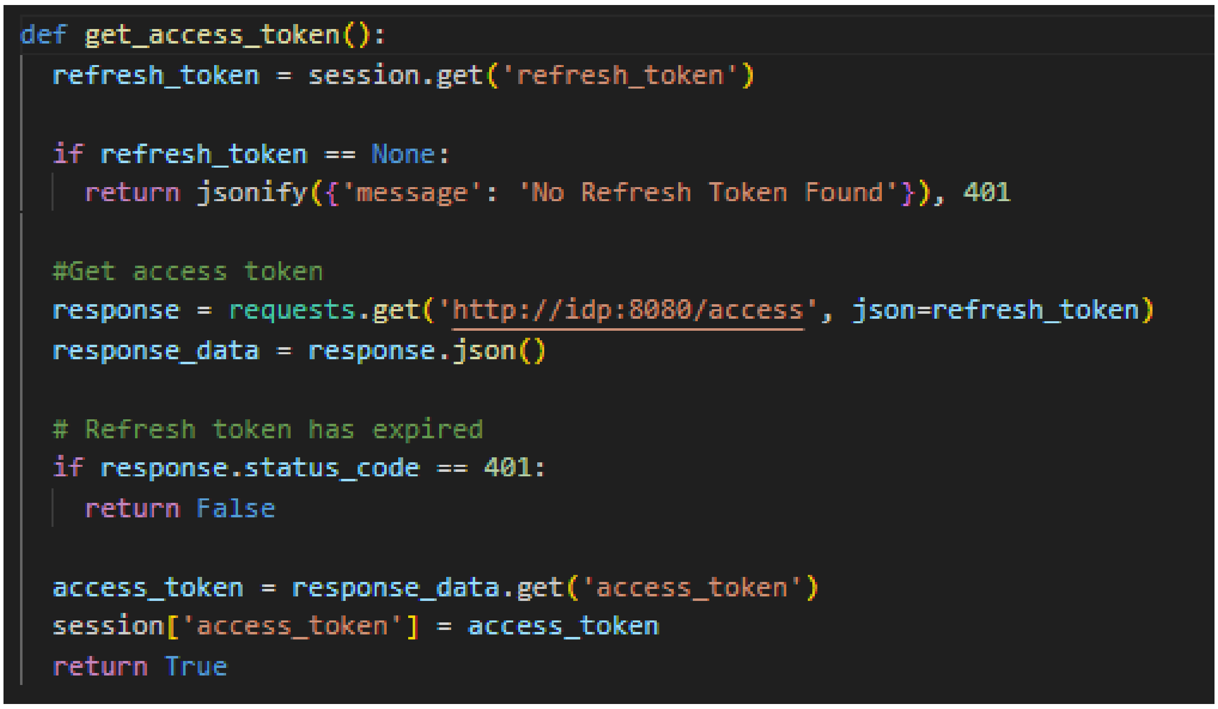
Every time a request is made to the resource server an access token is sent and the resource server validates it.



```python
@app.route('/verify', methods=['POST'])
def verify_token():
    # Should be the public key of the idp
    secret_key = "secret_key"
    access_token = request.headers.get('Authorization')
    try:
        payload = jwt.decode(access_token, secret_key, algorithms=['HS256'])
        # Check expiration time against current time
        current_time = datetime.now()
        if current_time >= datetime.~~utcfromtimestamp~~(payload['exp']):
            #raise error if expired
            raise jwt.ExpiredSignatureError
        return payload, 200
    except jwt.ExpiredSignatureError:
        return jsonify({'message': 'Expired access token'}), 403
    except jwt.InvalidTokenError:
        return jsonify({'message': 'Invalid access token'}), 401
    except Exception as e:
        return jsonify({'message': 'Error on access token'}), 401
```

Figure 22: Token Validation

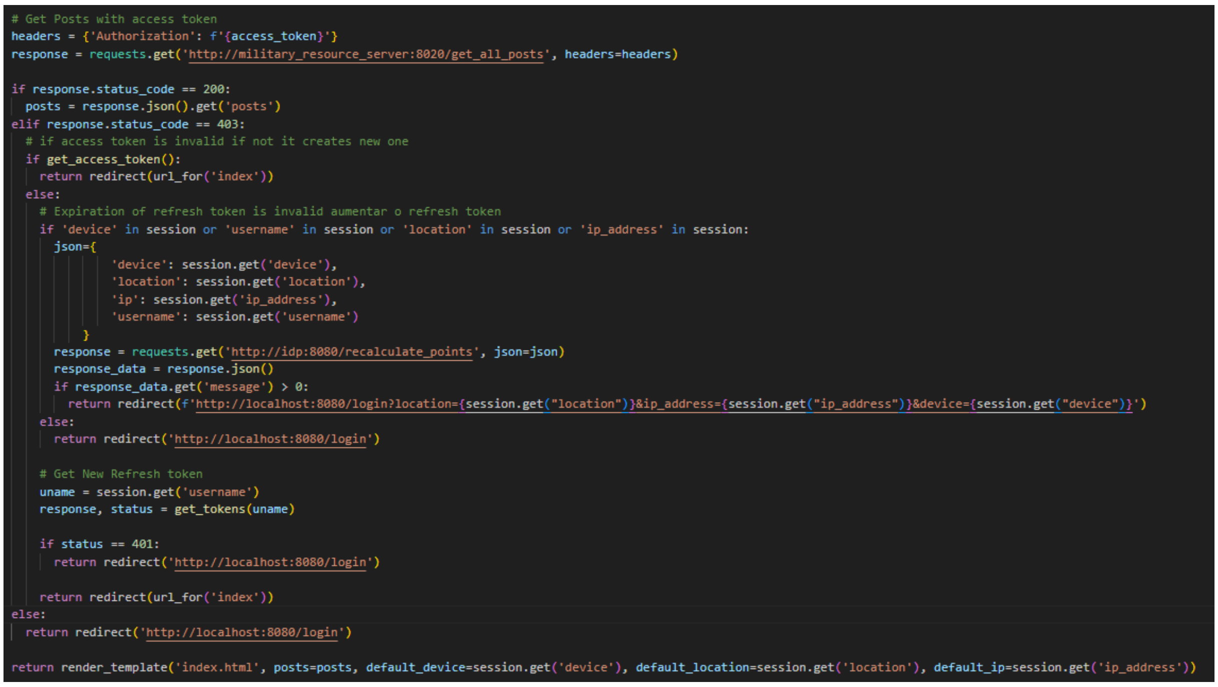
In case this above function returns 403 the application understands that the access token has expired, requesting a new access Token using the refresh token to do so.



```python
def get_access_token():
    refresh_token = session.get('refresh_token')

    if refresh_token == None:
        return jsonify({'message': 'No Refresh Token Found'}), 401

    #Get access token
    response = requests.get('http://idp:8080/access', json=refresh_token)
    response_data = response.json()

    # Refresh token has expired
    if response.status_code == 401:
        return False

    access_token = response_data.get('access_token')
    session['access_token'] = access_token
    return True
```

Figure 23: Request Access Token

However, if the Refresh token has also expired the application will first request the IDP to recalculate the session points, this function returns the number of MFA authentication methods the user needs to complete to pass authentication sending him back to the login page. In case he doesn't have MFA methods to pass his tokens are requested again.



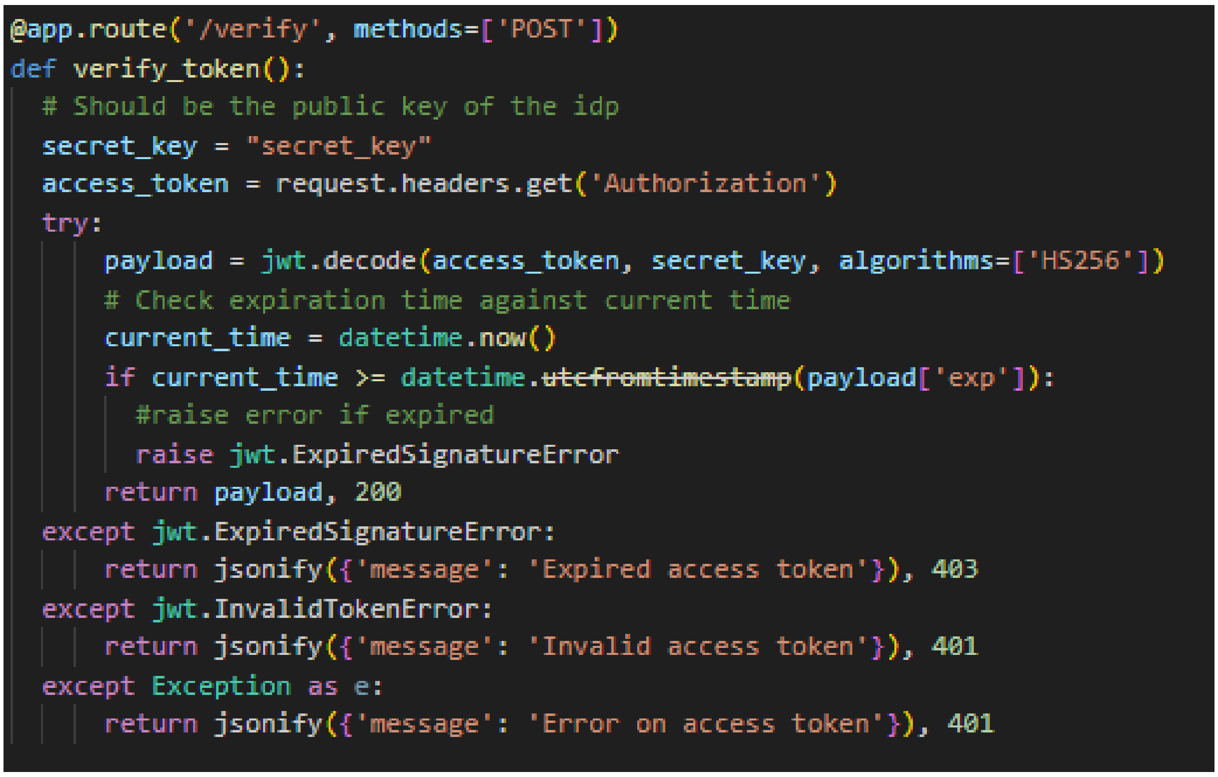
```python
# Get Posts with access token
headers = {'Authorization': f'{access_token}'}
response = requests.get('http://military_resource_server:8020/get_all_posts', headers=headers)

if response.status_code == 200:
  posts = response.json().get('posts')
elif response.status_code == 403:
  # if access token is invalid if not it creates new one
  if get_access_token():
    return redirect(url_for('index'))
  else:
    # Expiration of refresh token is invalid aumentar o refresh token
    if 'device' in session or 'username' in session or 'location' in session or 'ip_address' in session:
      json={
              'device': session.get('device'),
              'location': session.get('location'),
              'ip': session.get('ip_address'),
              'username': session.get('username')
          }
      response = requests.get('http://idp:8080/recalculate_points', json=json)
      response_data = response.json()
      if response_data.get('message') > 0:
        return redirect(f'http://localhost:8080/login?location={session.get("location")}&ip_address={session.get("ip_address")}&device={session.get("device")}')
    else:
      return redirect('http://localhost:8080/login')

    # Get New Refresh token
    uname = session.get('username')
    response, status = get_tokens(uname)

    if status == 401:
      return redirect('http://localhost:8080/login')

    return redirect(url_for('index'))
else:
  return redirect('http://localhost:8080/login')

return render_template('index.html', posts=posts, default_device=session.get('device'), default_location=session.get('location'), default_ip=session.get('ip_address'))
```

Figure 24: Request Refresh Token

3.6 Resource Server

As previously stated communications between the service and the resource server are done by HTTP with a cookie given by the IDP where the information about the user and their clearance is in the access token.

The Resource server needs then to verify the authenticity of the token and with the information stored on the token, decide the resources that said user has access to. First, we validate the token itself if it has expired we return ”403”, this code is important since the app uses it to understand if we need to request another token using the refresh token, or in case of any other error the user needs to re-authenticate, as explained on Token Expiration.

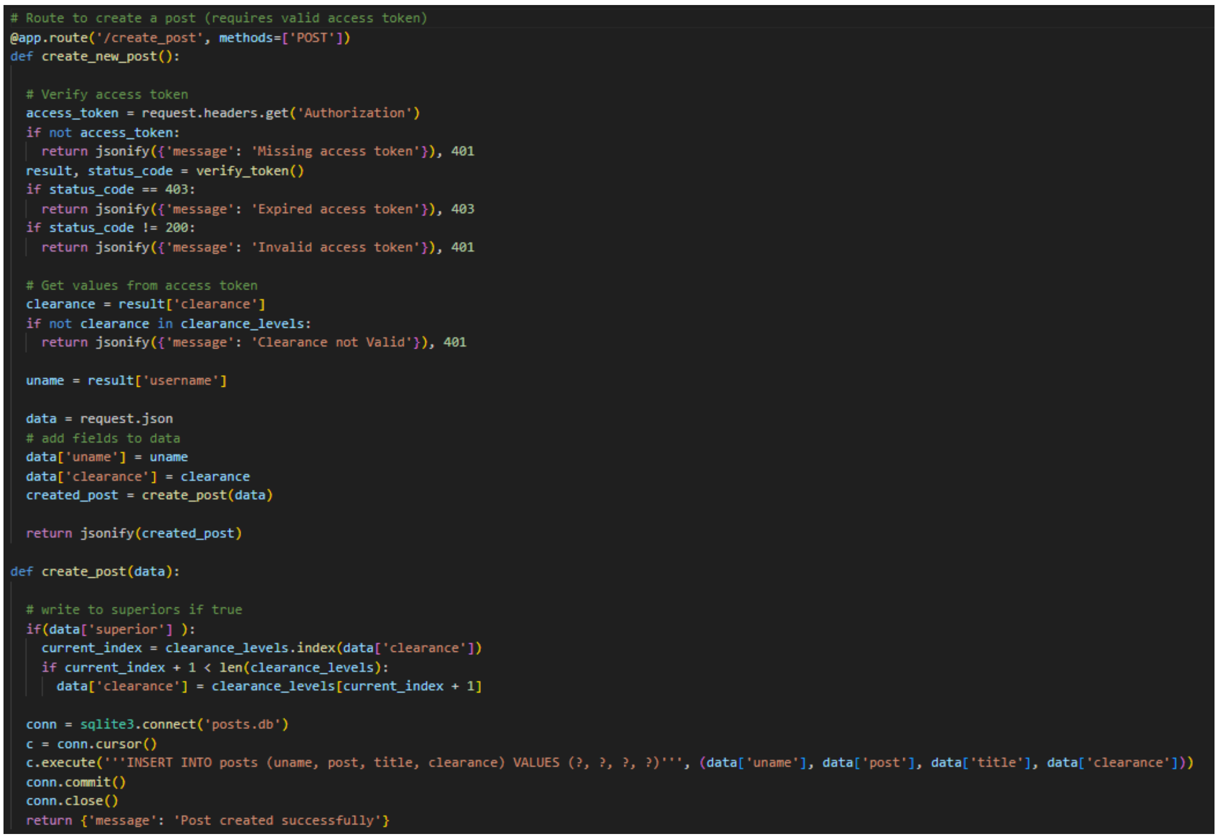


```
@app.route('/verify', methods=['POST'])
def verify_token():
    # Should be the public key of the idp
    secret_key = "secret_key"
    access_token = request.headers.get('Authorization')
    try:
        payload = jwt.decode(access_token, secret_key, algorithms=['HS256'])
        # Check expiration time against current time
        current_time = datetime.now()
        if current_time >= datetime.~~utcfromtimestamp~~(payload['exp']):
            #raise error if expired
            raise jwt.ExpiredSignatureError
        return payload, 200
    except jwt.ExpiredSignatureError:
        return jsonify({'message': 'Expired access token'}), 403
    except jwt.InvalidTokenError:
        return jsonify({'message': 'Invalid access token'}), 401
    except Exception as e:
        return jsonify({'message': 'Error on access token'}), 401
```

Figure 25: Token Validation

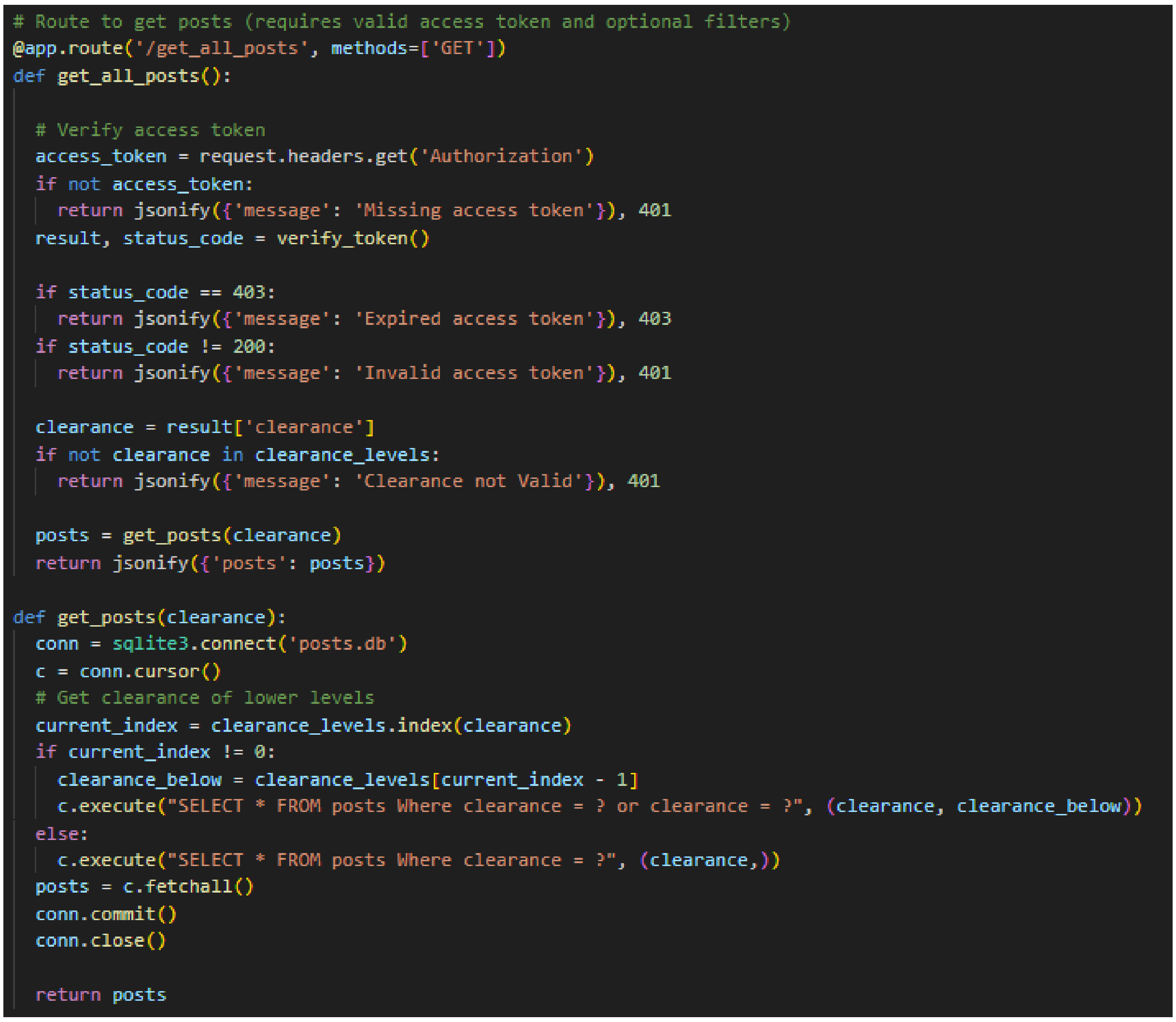
3.6.1 Military

As said in the previous report the first service is a military one based on the Bell–LaPadula model, this means that we can only see posts with a clearance level equal to or below our own and can only write to a clearance level one level above or equal to ours. To do this on the server there is a checkbox to indicate if the post that the user is writing should have a clearance level above his own.



```python
# Route to create a post (requires valid access token)
@app.route('/create_post', methods=['POST'])
def create_new_post():

  # Verify access token
  access_token = request.headers.get('Authorization')
  if not access_token:
    return jsonify({'message': 'Missing access token'}), 401
  result, status_code = verify_token()
  if status_code == 403:
    return jsonify({'message': 'Expired access token'}), 403
  if status_code != 200:
    return jsonify({'message': 'Invalid access token'}), 401

  # Get values from access token
  clearance = result['clearance']
  if not clearance in clearance_levels:
    return jsonify({'message': 'Clearance not Valid'}), 401

  uname = result['username']

  data = request.json
  # add fields to data
  data['uname'] = uname
  data['clearance'] = clearance
  created_post = create_post(data)

  return jsonify(created_post)

def create_post(data):

  # write to superiors if true
  if(data['superior'] ):
    current_index = clearance_levels.index(data['clearance'])
    if current_index + 1 < len(clearance_levels):
      data['clearance'] = clearance_levels[current_index + 1]

  conn = sqlite3.connect('posts.db')
  c = conn.cursor()
  c.execute('''INSERT INTO posts (uname, post, title, clearance) VALUES (?, ?, ?, ?)''', (data['uname'], data['post'], data['title'], data['clearance']))
  conn.commit()
  conn.close()
  return {'message': 'Post created successfully'}
```

Figure 26: Create Military Post

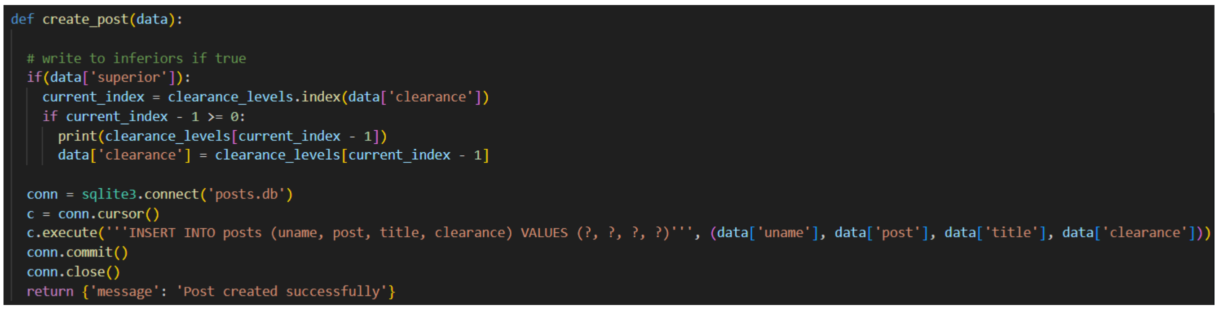


```python
# Route to get posts (requires valid access token and optional filters)
@app.route('/get_all_posts', methods=['GET'])
def get_all_posts():

    # Verify access token
    access_token = request.headers.get('Authorization')
    if not access_token:
        return jsonify({'message': 'Missing access token'}), 401
    result, status_code = verify_token()

    if status_code == 403:
        return jsonify({'message': 'Expired access token'}), 403
    if status_code != 200:
        return jsonify({'message': 'Invalid access token'}), 401

    clearance = result['clearance']
    if not clearance in clearance_levels:
        return jsonify({'message': 'Clearance not Valid'}), 401

    posts = get_posts(clearance)
    return jsonify({'posts': posts})

def get_posts(clearance):
    conn = sqlite3.connect('posts.db')
    c = conn.cursor()
    # Get clearance of lower levels
    current_index = clearance_levels.index(clearance)
    if current_index != 0:
        clearance_below = clearance_levels[current_index - 1]
        c.execute("SELECT * FROM posts Where clearance = ? or clearance = ?", (clearance, clearance_below))
    else:
        c.execute("SELECT * FROM posts Where clearance = ?", (clearance,))
    posts = c.fetchall()
    conn.commit()
    conn.close()

    return posts
```

Figure 27: Get post Military

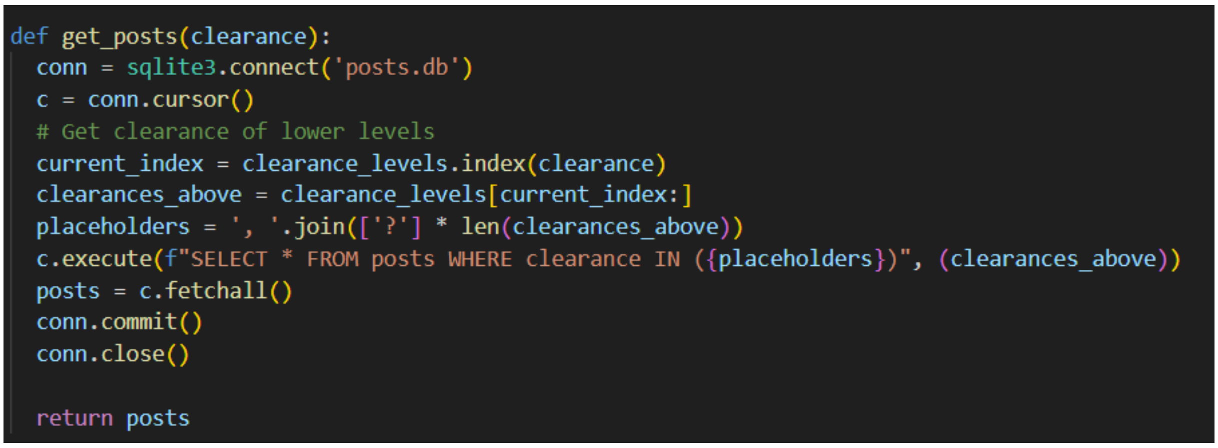
3.6.2 Religious

The Next service is Religious, based on the Biba model, this means that we can only see posts with an integrity level equal to or above our own and can only write to an integrity level one level below or equal to our own. To do this on the server there is a checkbox to indicate if the post that the user is writing should have a clearance level below.



```python
def create_post(data):

    # write to inferiors if true
    if(data['superior']):
        current_index = clearance_levels.index(data['clearance'])
        if current_index - 1 >= 0:
            print(clearance_levels[current_index - 1])
            data['clearance'] = clearance_levels[current_index - 1]

    conn = sqlite3.connect('posts.db')
    c = conn.cursor()
    c.execute('''INSERT INTO posts (uname, post, title, clearance) VALUES (?, ?, ?, ?)''', (data['uname'], data['post'], data['title'], data['clearance']))
    conn.commit()
    conn.close()
    return {'message': 'Post created successfully'}
```

Figure 28: Create Religious Post

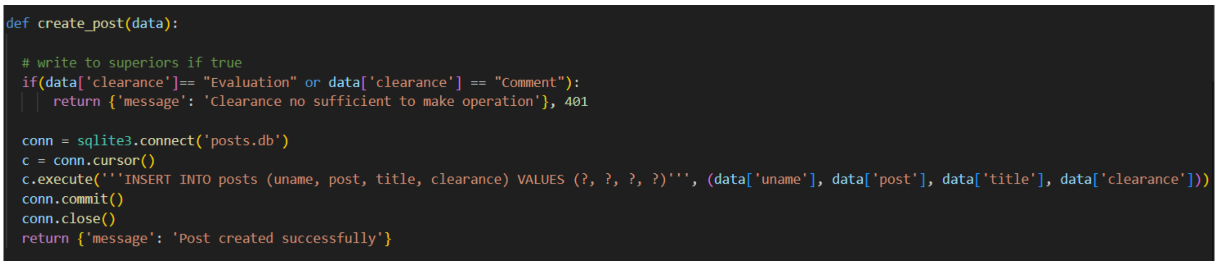


```python
def get_posts(clearance):
    conn = sqlite3.connect('posts.db')
    c = conn.cursor()
    # Get clearance of lower levels
    current_index = clearance_levels.index(clearance)
    clearances_above = clearance_levels[current_index:]
    placeholders = ', '.join(['?'] * len(clearances_above))
    c.execute(f"SELECT * FROM posts WHERE clearance IN ({placeholders})", (clearances_above))
    posts = c.fetchall()
    conn.commit()
    conn.close()

    return posts
```

Figure 29: Get post Religious

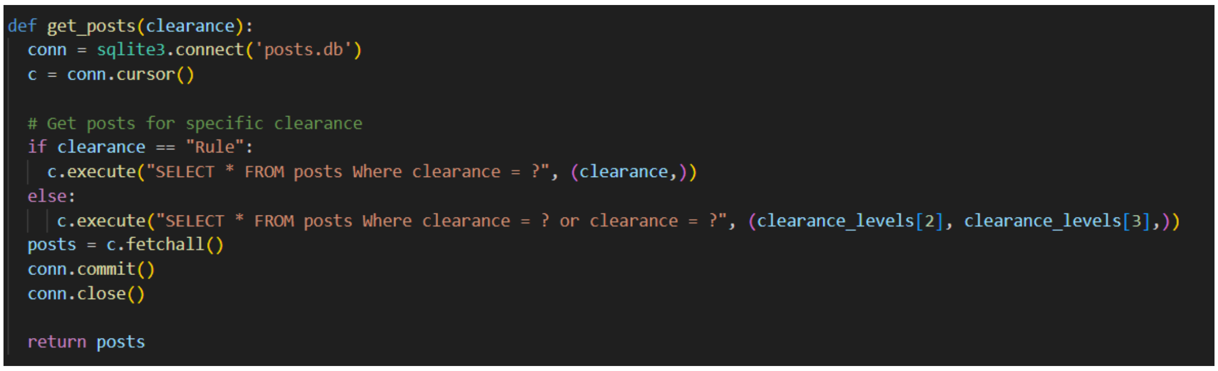
3.6.3 Cooking

The Final service is the cooking forum, based on the Biba model, on this service the two lower roles ”Looker” and ”Eater” can not create recipes and can only see posts above their role. The Cook Can Create recipes and see other recipes and the rules created by the Admin. Finally, the Admin can create Rules for their users and see other rules made by other Admins.



```python
def create_post(data):

    # write to superiors if true
    if(data['clearance']== "Evaluation" or data['clearance'] == "Comment"):
        return {'message': 'Clearance no sufficient to make operation'}, 401

    conn = sqlite3.connect('posts.db')
    c = conn.cursor()
    c.execute('''INSERT INTO posts (uname, post, title, clearance) VALUES (?, ?, ?, ?)''', (data['uname'], data['post'], data['title'], data['clearance']))
    conn.commit()
    conn.close()
    return {'message': 'Post created successfully'}
```

Figure 30: Create Cook post



```python
def get_posts(clearance):
    conn = sqlite3.connect('posts.db')
    c = conn.cursor()

    # Get posts for specific clearance
    if clearance == "Rule":
        c.execute("SELECT * FROM posts Where clearance = ?", (clearance,))
    else:
        c.execute("SELECT * FROM posts Where clearance = ? or clearance = ?", (clearance_levels[2], clearance_levels[3],))
    posts = c.fetchall()
    conn.commit()
    conn.close()

    return posts
```

Figure 31: Get Cook Post

4 Deployment and Testing

For easier deployment, we create a Docker-Compose with each service, resource_server, IDP, and mfa on different docker files.

All Flask application have their debug mode off and the deployed containers are running on the following URL:

- Services
  - Military: `http://localhost:8010`
  - Religious: `http://localhost:8011`
  - Cooking: `http://localhost:8012`
- Resource Server
  - Military: `http://localhost:8020`
  - Religious: `http://localhost:8021`
  - Cooking: `http://localhost:8022`
- IDP: `http://localhost:8080/login`
- Mfa
  - Email: `http://localhost:8001/email/<email>`
  - Phone: `http://localhost:8001/phone/<phone>`
  - TOTP: `http://localhost:8001/user/<username>`

And these are their endpoints:

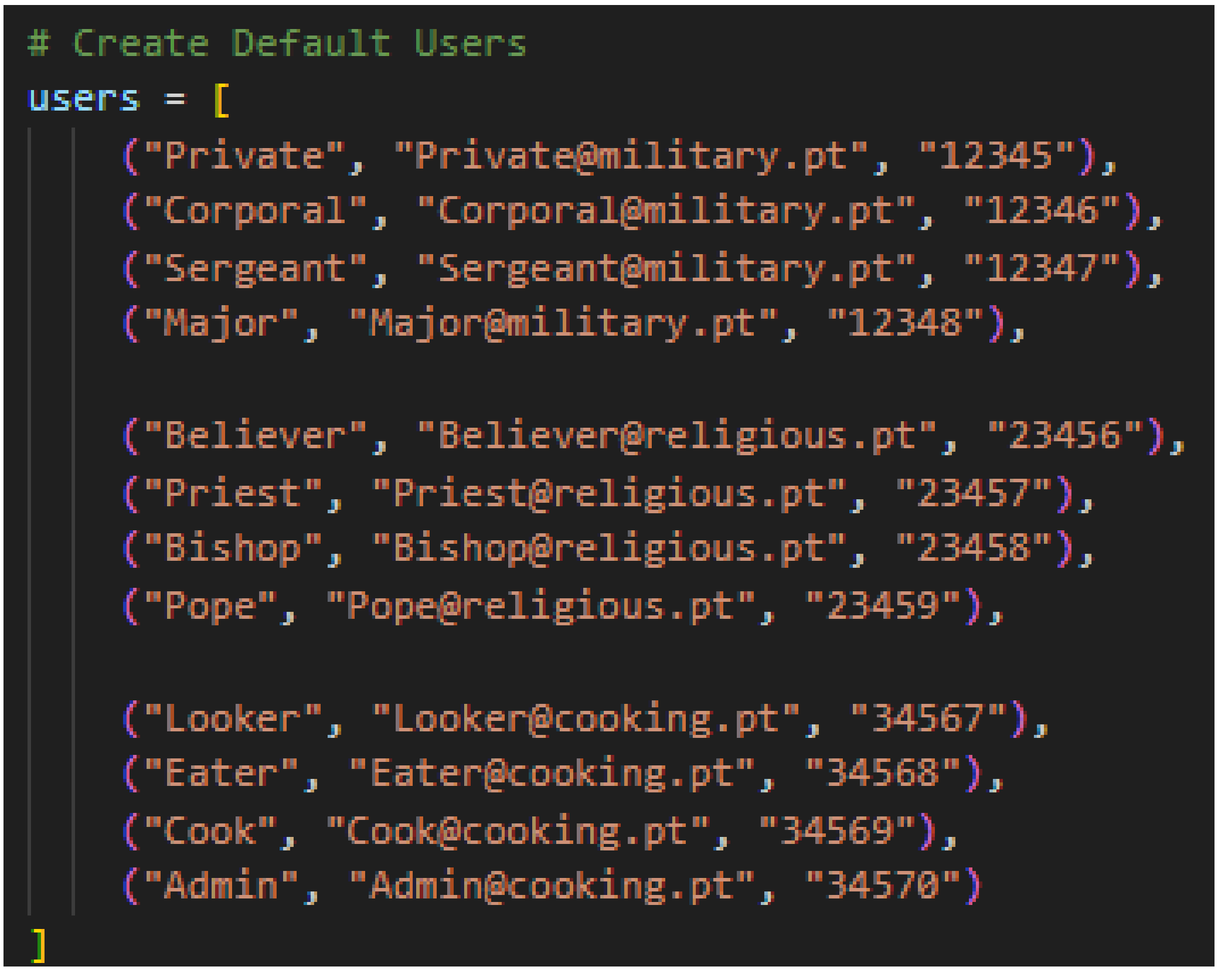
- **IDP:**
  - **"/login":** The endpoint used for getting forms to fill email/password and submitting them, it will also call functions related to calculating user points and implementing MFA.
  - **"/register":** Endpoint used registering new users.
  - **"/change_pass":** Endpoint used to change users password.
  - **"/Recalculate_points":** Used to calculate the points of a certain session.
  - **"/Authorization":** Used for creating a Auth Code based on json sent.
  - **"/Refresh":** Returns a refresh token based on Auth Code Provided.
  - **"/access":** Returns a Accesss token based on Refresh Token provided.

– **”/verify_sms/email/topt”:** Verifies if the code provided for sms/email/topt OTP is correct, then redirects for index, or for the next mfa method.

- **MFA:**
  – **”/phone/<Phone_num>”:** Shows the request code sent to the ”phone”
  – **”email/<Email>”:** Shows the request code sent to the ”email”
  – **”user/<user>”:** Shows the the TOPT

- **Services:**
  – **”/”:** The main page of the forum shows the posts that the user has access to.
  – **”/create”:** Endpoint with the html for creating posts.
  – **”/login”:** Redirects to the endpoint with the same name in the IDP.
  – **”/change_pass”:** Redirects to the endpoint with the same name in IDP.

To start testing simply access one of the Services URLs which will prompt you to login. For testing purposes, we have created default users with usernames and passwords that are identical.



```
# Create Default Users
users = [
        ("Private", "Private@military.pt", "12345"),
        ("Corporal", "Corporal@military.pt", "12346"),
        ("Sergeant", "Sergeant@military.pt", "12347"),
        ("Major", "Major@military.pt", "12348"),

        ("Believer", "Believer@religious.pt", "23456"),
        ("Priest", "Priest@religious.pt", "23457"),
        ("Bishop", "Bishop@religious.pt", "23458"),
        ("Pope", "Pope@religious.pt", "23459"),

        ("Looker", "Looker@cooking.pt", "34567"),
        ("Eater", "Eater@cooking.pt", "34568"),
        ("Cook", "Cook@cooking.pt", "34569"),
        ("Admin", "Admin@cooking.pt", "34570")
]
```

Figure 32: Default Users

After any verification code is asked you can access the corresponding URL to get the code or navigate to the terminal where it is also displayed.

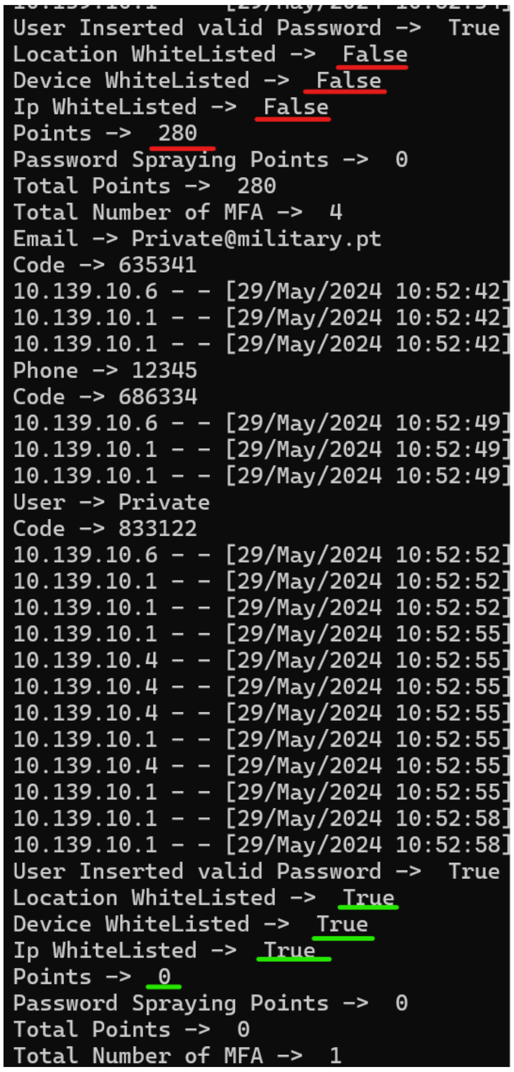
5 Results

Now we will demonstrate how the system authenticates users under various risk situations. This demonstration will focus on the most secure system, which is the military.

The First scenario will be a user authenticating with a **different device, IP and a new location**.

In the image below, we see that the user's device, IP address, and location were all different (not present in the whitelists). This giving the user 280 points, which on this service is equivalent to 4 methods of authentication: login, email, followed by SMS, and finally the totp.

The next login attempt with the same parameters, the score returned 0 as all the values where already added to the whiteList.



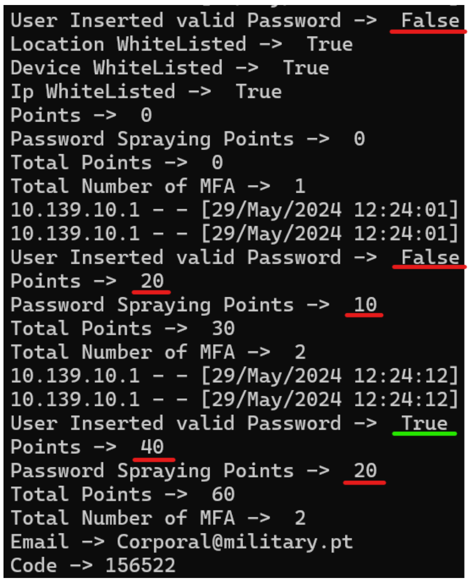
```
User Inserted valid Password ->  True
Location WhiteListed ->  False
Device WhiteListed ->  False
Ip WhiteListed ->  False
Points ->  280
Password Spraying Points ->  0
Total Points ->  280
Total Number of MFA ->  4
Email -> Private@military.pt
Code -> 635341
10.139.10.6 - - [29/May/2024 10:52:42]
10.139.10.1 - - [29/May/2024 10:52:42]
10.139.10.1 - - [29/May/2024 10:52:42]
Phone -> 12345
Code -> 686334
10.139.10.6 - - [29/May/2024 10:52:49]
10.139.10.1 - - [29/May/2024 10:52:49]
10.139.10.1 - - [29/May/2024 10:52:49]
User -> Private
Code -> 833122
10.139.10.6 - - [29/May/2024 10:52:52]
10.139.10.1 - - [29/May/2024 10:52:52]
10.139.10.1 - - [29/May/2024 10:52:52]
10.139.10.1 - - [29/May/2024 10:52:55]
10.139.10.4 - - [29/May/2024 10:52:55]
10.139.10.4 - - [29/May/2024 10:52:55]
10.139.10.4 - - [29/May/2024 10:52:55]
10.139.10.1 - - [29/May/2024 10:52:55]
10.139.10.4 - - [29/May/2024 10:52:55]
10.139.10.1 - - [29/May/2024 10:52:55]
10.139.10.1 - - [29/May/2024 10:52:58]
10.139.10.1 - - [29/May/2024 10:52:58]
User Inserted valid Password ->  True
Location WhiteListed ->  True
Device WhiteListed ->  True
Ip WhiteListed ->  True
Points ->  0
Password Spraying Points ->  0
Total Points ->  0
Total Number of MFA ->  1
```

Figure 33: Different Ip, Device and location

Now we will show how the system updates the score during a brute force attempt.

As shown below a user tried to login once which was **unsecesseful**. On the next unsuccessful attempt, the **score updated, adding 20** points for the brute force attempt and **10 points for the attempt made from the same IP** address. On the third time the user managed to insert the pass correctly but it already had **40 points** for the 2 times it got wrong **the credetials** and **20** for the 2 times the login **was unsuccessful from the same IP**.

Has we can see the scoring system worked has the mfa methods necessary to login went up by 1.



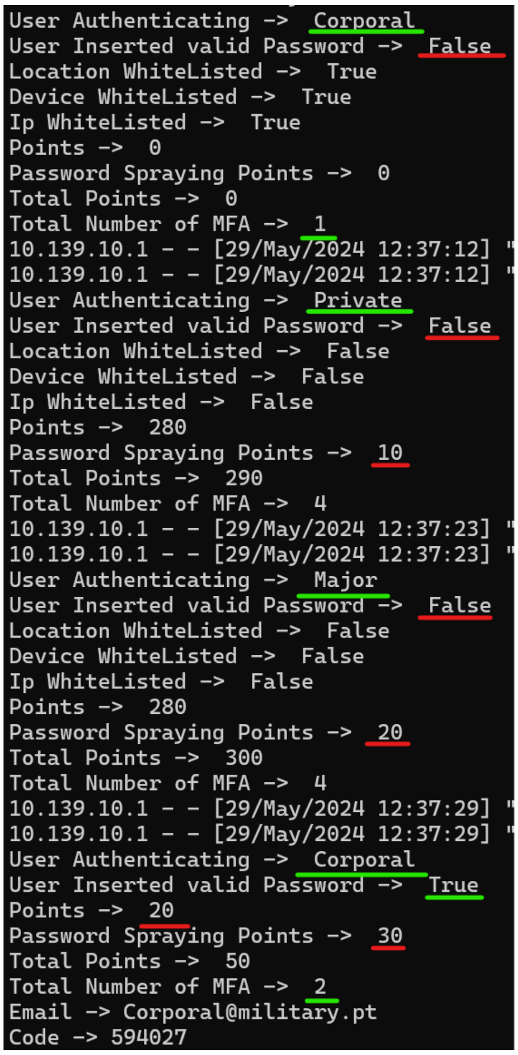
```
User Inserted valid Password ->  False
Location WhiteListed ->  True
Device WhiteListed ->  True
Ip WhiteListed ->  True
Points ->  0
Password Spraying Points ->  0
Total Points ->  0
Total Number of MFA ->  1
10.139.10.1 - - [29/May/2024 12:24:01]
10.139.10.1 - - [29/May/2024 12:24:01]
User Inserted valid Password ->  False
Points ->  20
Password Spraying Points ->  10
Total Points ->  30
Total Number of MFA ->  2
10.139.10.1 - - [29/May/2024 12:24:12]
10.139.10.1 - - [29/May/2024 12:24:12]
User Inserted valid Password ->  True
Points ->  40
Password Spraying Points ->  20
Total Points ->  60
Total Number of MFA ->  2
Email -> Corporal@military.pt
Code -> 156522
```

Figure 34: Brute Force Attempt

The following scenario we will show how the **Password spraying detection** works.

First we attempted to **unsuccessefuly login** with the users ”Corporal”, ”Private” and ”Major” using the **same IP** for each request. As we can see the points in each request **increased** by 10 points.

On the Final attempt we **sucessefuly** logged in the user ”Corporal” which had 20 points from the previous unsuccessful attempt and the 30 points from the 3 attempts with different users. This **increased** the mfa needed by 1.

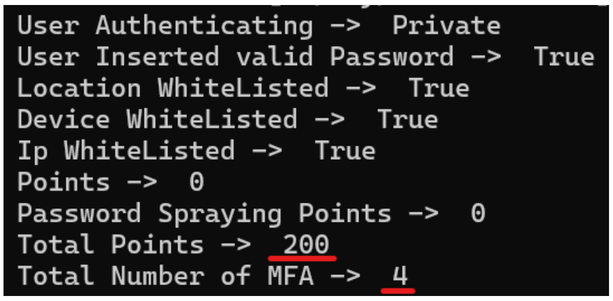


```
User Authenticating ->  Corporal
User Inserted valid Password ->  False
Location WhiteListed ->  True
Device WhiteListed ->  True
Ip WhiteListed ->  True
Points ->  0
Password Spraying Points ->  0
Total Points ->  0
Total Number of MFA ->  1
10.139.10.1 - - [29/May/2024 12:37:12] "
10.139.10.1 - - [29/May/2024 12:37:12] "
User Authenticating ->  Private
User Inserted valid Password ->  False
Location WhiteListed ->  False
Device WhiteListed ->  False
Ip WhiteListed ->  False
Points ->  280
Password Spraying Points ->  10
Total Points ->  290
Total Number of MFA ->  4
10.139.10.1 - - [29/May/2024 12:37:23] "
10.139.10.1 - - [29/May/2024 12:37:23] "
User Authenticating ->  Major
User Inserted valid Password ->  False
Location WhiteListed ->  False
Device WhiteListed ->  False
Ip WhiteListed ->  False
Points ->  280
Password Spraying Points ->  20
Total Points ->  300
Total Number of MFA ->  4
10.139.10.1 - - [29/May/2024 12:37:29] "
10.139.10.1 - - [29/May/2024 12:37:29] "
User Authenticating ->  Corporal
User Inserted valid Password ->  True
Points ->  20
Password Spraying Points ->  30
Total Points ->  50
Total Number of MFA ->  2
Email -> Corporal@military.pt
Code -> 594027
```

Figure 35: Password spraying detection

The last scenario involves a user attempting to change their password.

Has seen on the image below, the points for every parameter is 0 even the password spraying. However, **200 points were added** to the session, since the user is **trying to temper with it's credentials**, demanding the user to pass the 4 mfa methods.



```
User Authenticating ->  Private
User Inserted valid Password ->  True
Location WhiteListed ->  True
Device WhiteListed ->  True
Ip WhiteListed ->  True
Points ->  0
Password Spraying Points ->  0
Total Points ->  200
Total Number of MFA ->  4
```

Figure 36: Change Password

Our demonstration has shown how the system effectively handles various authentication scenarios, and that the scoring system ensures that attacks on the account trigger additional MFA methods, enhancing overall security. It needs to be noted that in our demonstration the number of authentication methods just went up by one since the simulation made the attacker guess the passwords with a small number of attempts, in a real scenario it would be probable that all 4 methods would be triggered since it would take a lot more tries to bruteforce the password.

6 Conclusion

In conclusion, through this project, we understood better the behavior of an Identity Provider. We also recognized its significance as a tool for centralizing account management and preventing attacks such as password spraying. By using centralized logging and monitoring of authentication, an IDP can more effectively detect and respond to suspicious activities across platforms, enhancing user security.