E2EE Messager documentation

By Olari Seppä

Table of contents

General description:

This web application focuses on authentication and end-to-end encrypted messaging between friends. Usage instructions are in README.MD of the GitHub repository.

Tools and technologies:

Frontend:

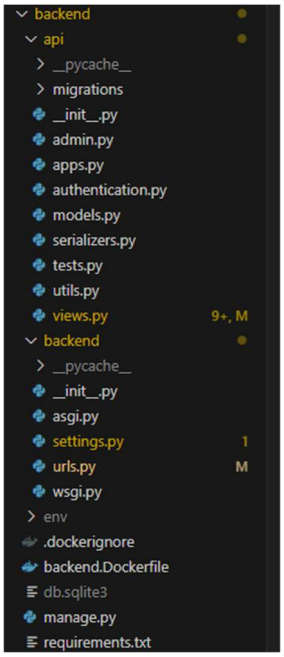
- TypeScript using React framework
- Libraries: React Component library MUI for easier UI building
- Web Crypto API for cryptographic operations

Backend:

- Python 3 using Django framework
- Django has great security by default and is well configurable for more safety features
- My used extra apps for Django: django-cors-headers, django rest framework, simplejwt token blacklist
- Multiple extra libraries
- Redis for caching

Containerization: Docker

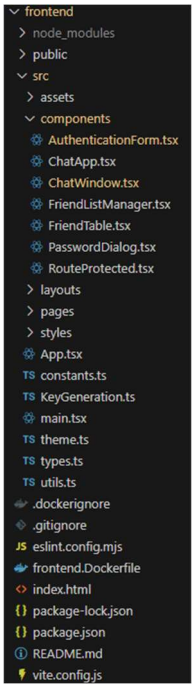
Program structure:



In the backend directory exists the Django project. Relevant files:

- models.py: Defines database models
- SQLite database
- views.py: Handle logic for http requests/responses
- settings.py: Settings (database, adding apps, middleware, csrf/cors settings)
- urls.py: URL routing
- authentication.py: Custom JWT authentication to use cookies (overwriting simple-jwt authentication method to use cookies)
- utils.py: Checking CSRF token function to call in views.py
- requirements.txt: All used libraries
- Dockerfile to build image

Frontend:



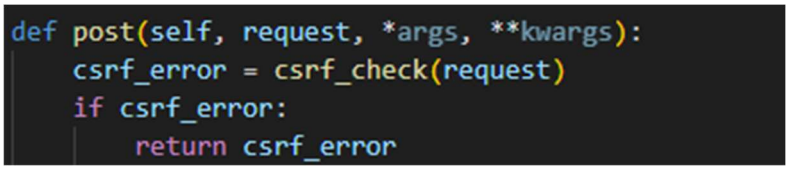
In the frontend directory exists the React project. Src contains the source code of the program, then there are components etc. Some relevant files:

- RouteProtected.tsx is for checking that the user is authenticated (has a valid JWT), and all components are wrapped with this component except for the login/register views to prevent unauthorized access.
- KeyGeneration.ts contains most of the functions concerning cryptography. These are then used inside components. More detail about this later.
- Dockerfile for frontend docker image
- package.json contains all the program dependencies

Other:

- .github/workflows/pipeline.yml: The CI pipeline code

Secure programming solutions:

- Requiring CSRF tokens for any POST/PUT/DELETE requests
  - Important for cookie-based authentication
  - Using Django CsrfViewMiddleware, custom function had to be implemented (Django does this automatically with session authentication but not in the case of using cookies)
  - 
    ```
    def csrf_check(request):
        csrf_middleware = CsrfViewMiddleware(lambda req: None)
        reason = csrf_middleware.process_view(request, None, (), {})
        if reason:
            return Response({"detail": f"CSRF Failed: {reason}"}, status=403)
        return None
    ```
  - For any POST/PUT/DELETE: check csrf token
  - 
    ```
    def post(self, request, *args, **kwargs):
        csrf_error = csrf_check(request)
        if csrf_error:
            return csrf_error
    ```

- Error and exception handling
  - Avoid leaking information
- Authentication:
  - JWT stored in HTTPonly cookies so not vulnerable to JavaScript

```
            # Authentication
            response = super().post(request, *args, **kwargs)

            if response.status_code == 200:
                tokens = response.data

                response.set_cookie(
                    key="access_token",
                    value=tokens["access"],
                    httponly=True,
                    secure=True,
                    samesite="Lax",  # Strict?
                )

                response.set_cookie(
                    key="refresh_token",
                    value=tokens["refresh"],
                    httponly=True,
                    secure=True,
                    samesite="Lax",
                )
            return response
```

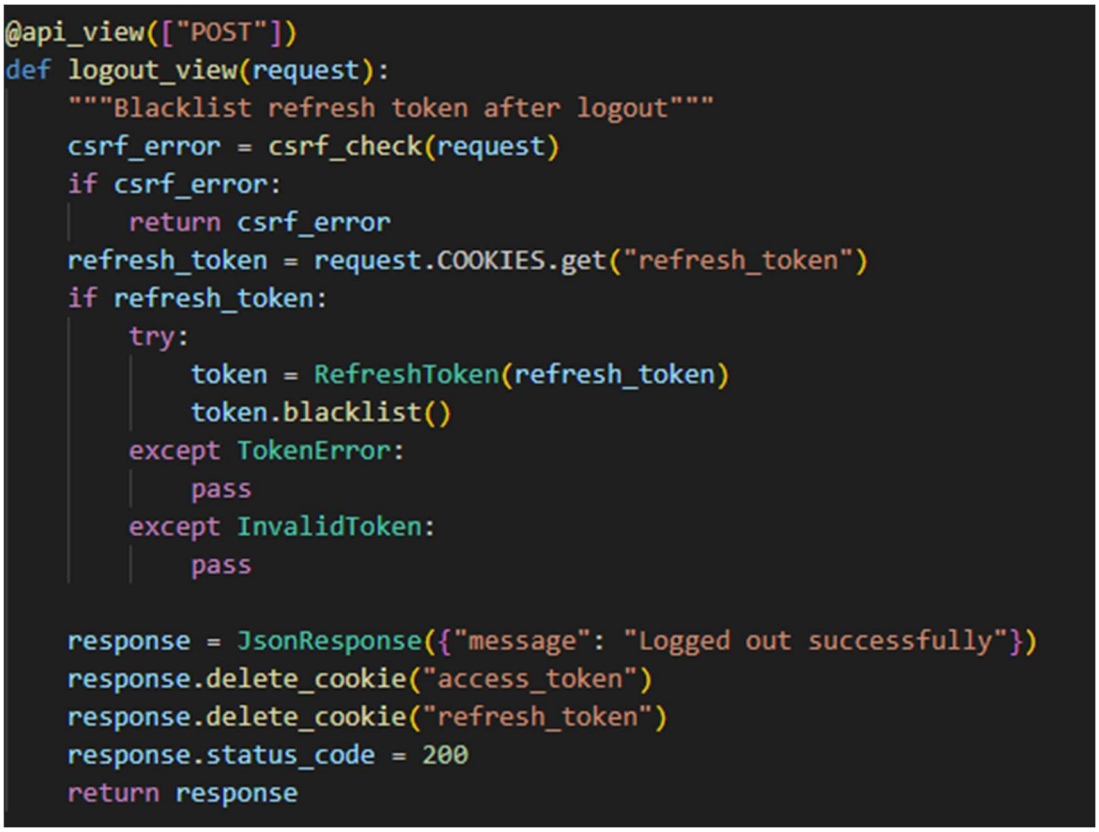
-

```
class CustomTokenRefreshView(TokenRefreshView):
    permission_classes = [AllowAny]

    def post(self, request, *args, **kwargs):
        refresh_token = request.COOKIES.get("refresh_token")
        if not refresh_token:
            return Response(
                {"detail": "Missing refresh token"}, status=status.HTTP_400_BAD_REQUEST
            )
        try:
            refresh = RefreshToken(refresh_token)
            access_token = str(refresh.access_token)

            response = Response({"Message": "Token refreshed"})
            response.set_cookie(
                key="access_token",
                value=access_token,
                httponly=True,
                secure=True,  # true for production
                samesite="Lax",
            )
            return response
        except TokenError:
            return Response({"detail": "Invalid refresh token"}, status=401)
```

- used JWTs get blacklisted after logging out



```python
@api_view(["POST"])
def logout_view(request):
    """Blacklist refresh token after logout"""
    csrf_error = csrf_check(request)
    if csrf_error:
        return csrf_error
    refresh_token = request.COOKIES.get("refresh_token")
    if refresh_token:
        try:
            token = RefreshToken(refresh_token)
            token.blacklist()
        except TokenError:
            pass
        except InvalidToken:
            pass

    response = JsonResponse({"message": "Logged out successfully"})
    response.delete_cookie("access_token")
    response.delete_cookie("refresh_token")
    response.status_code = 200
    return response
```

- Access tokens are kept with a short lifespan
- Password validation when registering new users (length, similarity, cannot be numeric, a list of commonly used passwords is blacklisted). Pretty much built in to Django, just add validators to settings.py
- Progressive login/register throttling (progressive ban lengths for too many attempts).

```python
r = redis.Redis(host="redis", port=6379, db=0)
User = get_user_model()


def handle_violation(request):
    """Progressive rate limit for failed login attempts"""

    ip = request.META.get("REMOTE_ADDR")
    key = f"rate_violation:{ip}"
    count = r.incr(key)
    r.expire(key, 3600)  # violation expires in one hour

    if count == 1:
        r.setex(f"ban:{ip}", 60, 1)  # first block for 1 min

    elif count == 2:
        r.setex(f"ban:{ip}", 600, 1)  # 2nd block for 10 mins

    elif count >= 3:
        r.setex(f"ban:{ip}", 1800, 1)  # 3rd block for 30 mins
```

```python
    @method_decorator(ratelimit(key="ip", rate="5/m", block=False))
    def post(self, request, *args, **kwargs):
        # Check if person is blocked due to too many login attempts
        ip = request.META.get("REMOTE_ADDR")
        if r.exists(f"ban:{ip}"):
            duration = r.expiretime(f"ban:{ip}") - int(time.time())
            return Response(
                {
                    "detail": f"Temporarily banned: too many login attempts, {duration} sec
                },
                status=429,
            )
        # Check if person exceeded the rate limit and handle the violation
        if getattr(request, "limited", "false"):
            handle_violation(request)
            return Response({"detail": "Too many login attempts!"}, status=429)
```

- Login attempts are logged in the database
- Django admin page url is changed and also rate limited
- Django handles password hashing, I switched to argon2id instead of default PBKDF2. (added argon2id library and modified hashing in settings.py, easy process.) Although Djangos default PBKDF2 is very sufficient with 870 000 iterations according to OWASP password storage cheat sheet https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html

End-to-end encryption:

Used key exchange algorithm is Elliptic Curve Diffie-Hellman (ECDH) with an elliptic curve NIST p-256. There are possibly better algorithms available like X25519. Let's say that we have Alice and Bob trying to message each other. Both generate a keypair for themselves, consisting of a private and public key. The public keys are fine to share with each other and private keys must be kept a secret. The way Alice and Bob can communicate with each other is that they exchange their public keys and then compute a shared key using their own private key and the other's public key. Both will then have the same shared key. This shared key is then used as a symmetrical key to encrypt and decrypt messages. The used encryption algorithm for messages (and private key) in this program is AES-GCM. AES-GCM provides confidentiality, integrity and authenticity (prevents message tampering).

Implementation in this program:

This program uses Web Crypto API for all cryptographic functions.

https://developer.mozilla.org/en-US/docs/Web/API/Crypto

Salts and initialization vectors are generated using crypto.getRandomValues() which should be cryptographically strong. Used ivs are 96-bit and salts 128-bit which aligns with NIST recommendations. Messages between users are stored in the database along with the ivs. Only the sender and the receiver can be figured out, the content is encrypted.

Initial key generation process for users: Each users public key will be saved in the backend database associated with the user. This is fine, as a public key alone is useless for an attacker.

```typescript
// Generate initial keypair for the user, only once
export async function InitialUserKeyGeneration(userId: number, username: string, password: string) {
  // https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/generateKey
  const keyPair = await window.crypto.subtle.generateKey(
    {
      name: "ECDH",
      namedCurve: "P-256",
    },
    true,
    ["deriveKey"]
  );

  // create indexedDB storage
  try {
    const db = await openDB('keys-db', 1, {
      upgrade(db) {
        db.createObjectStore('keys')
      }
    })
    // Get private key in jwk format
    const privateKeyJwk = await crypto.subtle.exportKey('jwk', keyPair.privateKey)

    // Encrypt the private key with user given password
    const encryptedPrivateKeyJwk = await encryptPrivateKey(privateKeyJwk, password)

    // Put the encrypted key in indexedDB
    await db.put('keys', {
      ciphertext: Array.from(encryptedPrivateKeyJwk.ciphertext),
      iv: Array.from(encryptedPrivateKeyJwk.iv),
      salt: Array.from(encryptedPrivateKeyJwk.salt)
    }, `privatekey-${userId}-${username}`)

  } catch (error) {
    console.log(error)
  }
```

Generating keypair and storing private key to indexedDB, first encrypting it. Public key is also sent to backend associated to the user after this.

The private key storage is different: it is only stored client-side, in this case the browsers indexedDB is the choice. It is not ideal, but it is the best option for a purely web app. To increase security, the private key is first encrypted with AES-GCM before putting it in indexedDB. This is done with a user-provided password. From the password, encryption key is derived using PBKDF2. The password is asked again when accessing messages (to get the private key from indexedDB). If a private key is not encrypted inside indexedDB, it can be easily accessed by XSS or malware on the users computer.

Some functions using web crypto API:

```
178   // Generate shared secret key from senders private key and receivers public key
179 export async function genSharedKey(senderPrivKey: CryptoKey, receiverPubKey: CryptoKey) {
180   if (!senderPrivKey || !receiverPubKey) {
181     console.log("Error: senderPrivKey or receiverPubKey is null")
182     return
183   }
184   // Derive a shared secret key using ECDH
185   // https://developer.mozilla.org/en-US/docs/Web/API/SubtleCrypto/deriveKey
186   const sharedSecret = await crypto.subtle.deriveKey(
187     {
188       name: 'ECDH',
189       public: receiverPubKey,
190     },
191     senderPrivKey,
192     {
193       name: 'AES-GCM',
194       length: 256,
195     },
196     true,
197     ['encrypt', 'decrypt']
198   )
199
200   return sharedSecret
201 }
```

Generation of a shared key with senders private key and receivers public key

```
214 // Function for decrypting the messages with secret shared key
215 export async function decryptMessages(messages: Message[], sharedKey: CryptoKey) {
216   // For debugging
217   // const rawKey = await crypto.subtle.exportKey("raw", sharedKey);
218   // console.log(arrayBufferToBase64(rawKey))
219   const dec = new TextDecoder()
220   const decryptedMessages = await Promise.all(
221     messages.map(async (msg) => {
222       const encryptedBuffer = base64ToArrayBuffer(msg.content)
223       const ivBuffer = base64ToArrayBuffer(msg.iv)
224       const decrypted = await crypto.subtle.decrypt(
225         {
226           name: 'AES-GCM',
227           iv: ivBuffer
228         },
229         sharedKey,
230         encryptedBuffer
231       )
232       msg.content = dec.decode(decrypted)
233       return msg
234     }))
235
236   return decryptedMessages
237 }
```

Decrypting messages with the shared key

```
34
35  // Encrypt a private key (JWK form) before storing it
36  export async function encryptPrivateKey(privJwk: JsonWebKey, password: string) {
37    const enc = new TextEncoder()
38    const iv = crypto.getRandomValues(new Uint8Array(12)) // 96 bit iv
39    const salt = crypto.getRandomValues(new Uint8Array(16)) // 128 bit salt
40
41    const key = await deriveKeyFromPassword(password, salt)
42
43    const data = enc.encode(JSON.stringify(privJwk))
44    const encrypted = await crypto.subtle.encrypt(
45      { name: 'AES-GCM', iv: iv },
46      key,
47      data,
48    )
49
50    return {
51      ciphertext: new Uint8Array(encrypted),
52      iv,
53      salt
54    }
55  }
56
```

Encrypting a private key

```
6   // Derive encryption key from a password with PBKDF2
7   export async function deriveKeyFromPassword(password: string, salt: Uint8Array): Promise<CryptoKey> {
8     const enc = new TextEncoder()
9     const passwordKey = await crypto.subtle.importKey(
10      'raw',
11      enc.encode(password),
12      { name: "PBKDF2" },
13      false,
14      ["deriveKey"]
15    )
16    // https://developer.mozilla.org/en-US/docs/Web/API/Pbkdf2Params
17    // https://developer.mozilla.org/en-US/docs/Web/API/AesKeyGenParams
18    return await crypto.subtle.deriveKey(
19      {
20        name: 'PBKDF2',
21        salt: salt,
22        iterations: 600_000,
23        hash: 'SHA-256'
24      },
25      passwordKey,
26      {
27        name: 'AES-GCM',
28        length: 256
29      },
30      true,
31      ['encrypt', 'decrypt']
32    )
33  }
34
```

Deriving encryption key with PBKDF2

Testing:

Manual testing

Test case 1 - Tampering the message ciphertext. Expected result: Message fails to decrypt as integrity is broken. Result: pass

Test case 2 – Encrypted message is not the same even if the msg is same. Result: pass

Test case 3 – Logout, then manually set cookie refresh token to the one that previously existed and try to log in. Expected: 401 due to blacklist. Result: pass

Test case 4 – Attempt to make API calls without logging in. Expected: 401. Result: pass

Test case 5 – Attempt to exploit API calls while logged in. Expected: not possible. Result: Found vulnerable API endpoint: updating user info. This endpoint is used to update the authenticated users E2EE public key. By bypassing the frontend they can set this to any value. The person might set their public key identical to another persons. In this case, if a third person is messaging with the two people with identical public keys, their messages will be using the same shared key which is a big risk for impersonation or MITM attacks.

Fixed: making sure that everyone has a unique public key with the following code:

```
class User(AbstractUser):
    e2ee_public_key = models.JSONField(unique=True, null=True, blank=True)

    def clean(self):
        super().clean()
        if self.e2ee_public_key:
            x = self.e2ee_public_key.get("x")
            y = self.e2ee_public_key.get("y")
            crv = self.e2ee_public_key.get("crv")

            if not (x and y and crv):
                raise ValidationError("Invalid public key")

            # Check for duplicate public keys
            for user in User.objects.exclude(pk=self.pk).iterator():
                key = user.e2ee_public_key
                if not key:
                    continue
                if key.get("x") == x and key.get("y") == y and key.get("crv") == crv:
                    raise ValidationError("Public key already in use")
```

Automatic testing

Throughout about half of the project a GitHub Actions self-hosted runner was used to run a CI pipeline on every commit. The pipeline included the following security scanning tools: Snyk, Semgrep, Trivy (filesystem and container scan) and ZAP (Zed Attack Proxy). The full pipeline code can be found in the earlier mentioned .github/workflows/pipeline.yml. Artifacts of the reports were stored.

Snyk:

Found vulnerabilities in react-router-dom 7.5.0. https://security.snyk.io/vuln/SNYK-JS-REACTROUTER-9804426

https://security.snyk.io/vuln/SNYK-JS-REACTROUTER-9804420

These vulnerabilities actually appeared during the project. CVSS scores were high at 8.7. Fixed by updating to version 7.5.2.

On backend, a medium 5.5 vulnerability on djangorestframework-simplejwt plugin. This is regarding inactive users being able to gain unauthorized access when using custom authentication logic and not validating the user. My program doesn't handle inactive users so I will not fix this issue.

Semgrep:

1. Detected data rendered directly to the end user via 'HttpResponse' or a similar object. This bypasses Django's built-in cross-site scripting (XSS) defenses and could result in an XSS vulnerability. Use Django's template engine to safely render HTML.

I was just rendering simple raw string responses if ratelimit happens on the Django admin panel, which should be safe. Switched to JSONResponse instead to avoid the warning

2. Django REST framework configuration is missing default rate- limiting options. This could inadvertently allow resource starvation or Denial of Service (DoS) attacks. Add 'DEFAULT_THROTTLE_CLASSES' and 'DEFAULT_THROTTLE_RATES' to add rate-limiting to your application.

I am using django-ratelimit but only for login/registering. Ideally I would rate limit every API endpoint

3. By not specifying a USER, a program in the container may run as 'root'. This is a security hazard. If an attacker can control a process running as root, they may have control over the container. Ensure that the last USER in a Dockerfile is a USER other than 'root'.

Fix: Adding a user to docker container and running the program as user instead of root. I am not implementing this fix at this moment due to running into permission issues regarding database and I want to make sure that the app runs on the exercise works reviewer's computer.

Trivy:

Filescan:

```
frontend/package-lock.json (npm)
================================
Total: 1 (UNKNOWN: 0, LOW: 0, MEDIUM: 1, HIGH: 0, CRITICAL: 0)

┌─────────┬─────────────────┬──────────┬────────┬───────────────────┬──────────────────────────────┬──────────────────────────────────────────────────────────────┐
│ Library │ Vulnerability   │ Severity │ Status │ Installed Version │        Fixed Version         │                            Title                             │
├─────────┼─────────────────┼──────────┼────────┼───────────────────┼──────────────────────────────┼──────────────────────────────────────────────────────────────┤
│ vite    │ CVE-2025-46565 │ MEDIUM   │ fixed  │ 6.2.6             │ 6.3.4, 6.2.7, 6.1.6, 5.4.19, 4.5.14 │ vite: Path Traversal in Vite Dev Server Allows Access to │
│         │                 │          │        │                   │                              │ Restricted Files...                                          │
│         │                 │          │        │                   │                              │ https://avd.aquasec.com/nvd/cve-2025-46565                   │
└─────────┴─────────────────┴──────────┴────────┴───────────────────┴──────────────────────────────┴──────────────────────────────────────────────────────────────┘
```

Vulnerability published on 01.05.2025, fixed with new version of Vite

```
env/lib/python3.12/site-packages/sslserver/certs/development.key (secrets)
===========================================================================
Total: 1 (UNKNOWN: 0, LOW: 0, MEDIUM: 0, HIGH: 1, CRITICAL: 0)

HIGH: AsymmetricPrivateKey (private-key)
================================================
Asymmetric Private Key
------------------------------------------------
 env/lib/python3.12/site-packages/sslserver/certs/development.key:1
```

This is related to django-sslserver that I temporarily used to run the app on HTTPS for local network testing purposes with multiple computers. The private key is exposed in the filesystem but it is not in real use so it doesn't matter this time.

Container scan:

```
secprog300_backend (debian 12.10)
=================================
Total: 104 (UNKNOWN: 0, LOW: 72, MEDIUM: 30, HIGH: 1, CRITICAL: 1)
```

```
secprog300_frontend (debian 12.10)
==================================
Total: 1343 (UNKNOWN: 9, LOW: 690, MEDIUM: 551, HIGH: 88, CRITICAL: 5)
```

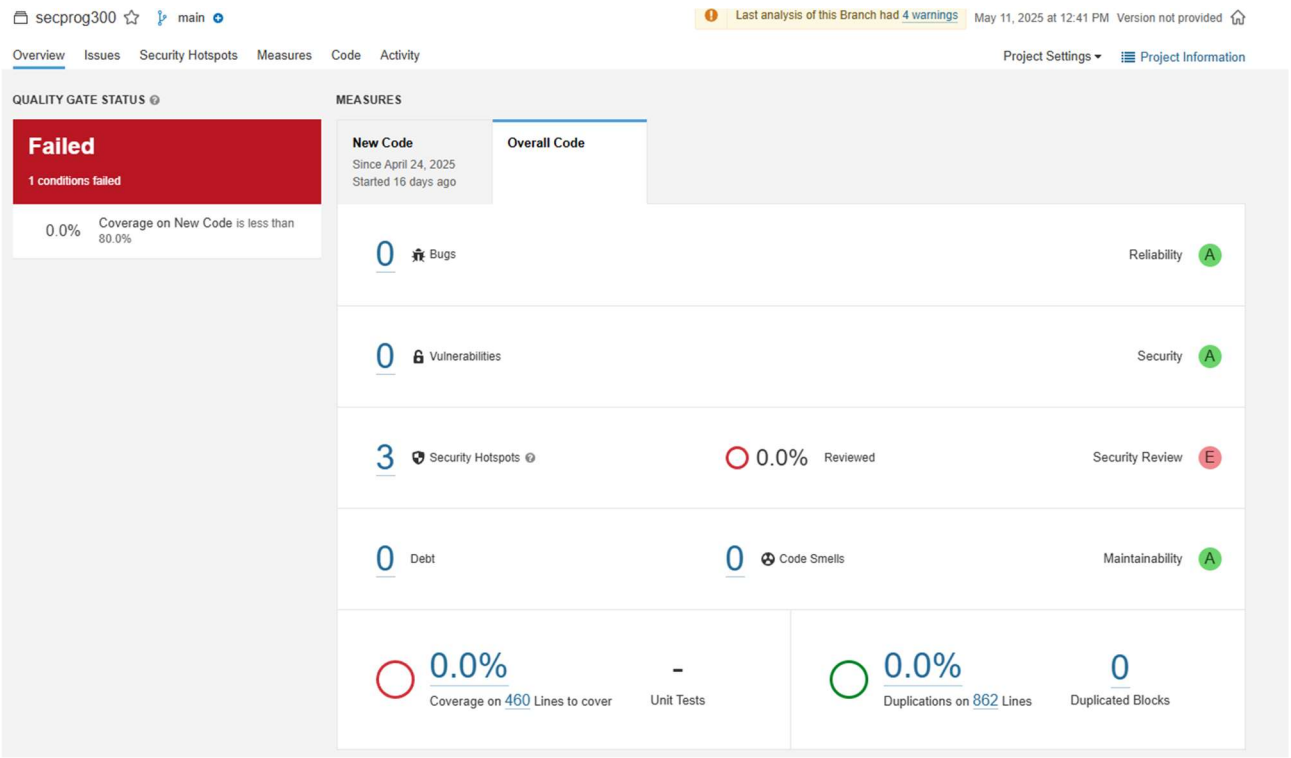
```
redis:7 (debian 12.10)
======================
Total: 76 (UNKNOWN: 0, LOW: 59, MEDIUM: 15, HIGH: 1, CRITICAL: 1)
```

Most of these are not affecting my project as the dependencies having the vulnerability are not used. I tried to update the image versions to ones with less vulnerabilities though. Here are the results after:

```
secprog300_backend (debian 12.10)
=================================
Total: 80 (UNKNOWN: 0, LOW: 60, MEDIUM: 18, HIGH: 1, CRITICAL: 1)
```

```
secprog300_frontend (debian 12.10)
==================================
Total: 75 (UNKNOWN: 0, LOW: 59, MEDIUM: 14, HIGH: 1, CRITICAL: 1)
```

SonarQube:



The test coverage for this project is 0% so SonarQube complains about that.

Security hotspots:

```
def logout_view(request):
```

Make sure allowing safe and unsafe HTTP methods is safe here. Comment

```
    """Blacklist refresh token after logout"""
    csrf_error = csrf_check(request)
    if csrf_error:
        return csrf_error
    refresh_token = request.COOKIES.get("refresh_token")
    if refresh_token:
        try:
            token = RefreshToken(refresh_token)
            token.blacklist()
```

The Django logout view doesn't restrict the http method, fixed by allowing POST only. GET method would disable the CSRF check here.

Another thing Sonarqube detected was that my Django settings had debug mode on which is unsuitable for production.

ZAP:

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	2
Low	3
Informational	2
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	3
Missing Anti-clickjacking Header	Medium	3
Insufficient Site Isolation Against Spectre Vulnerability	Low	9
Permissions Policy Header Not Set	Low	5
X-Content-Type-Options Header Missing	Low	6
Modern Web Application	Informational	3
Storable but Non-Cacheable Content	Informational	6

On the backend:

Summary of Alerts

Risk Level	Number of Alerts
High	0
Medium	1
Low	2
Informational	1
False Positives:	0

Alerts

Name	Risk Level	Number of Instances
Content Security Policy (CSP) Header Not Set	Medium	3
Permissions Policy Header Not Set	Low	3
Server Leaks Version Information via "Server" HTTP Response Header Field	Low	3
Storable and Cacheable Content	Informational	3

The highest found vulnerability was medium risk. Content security policy could be set with django-csp plugin.

Owasp top 10

How the program deals with possible issues:

1. **Broken access control:**

- Require authentication for API endpoints
- ```
  @api_view(["POST"])
  @permission_classes([IsAuthenticated])
  def delete_friend(request):
  ```
- This view allows only POST request and requires user to be authenticated
- Allowing CORS only for frontend
- API endpoints for getting user information do not allow any user ID as params etc.
- ```
  @method_decorator(ensure_csrf_cookie, name="get")
  class CheckAuthenticationView(APIView):
      # check if authenticated and get user information
      permission_classes = [IsAuthenticated]

      def get(self, request):
          user = request.user
          return Response(
              {"name": user.get_username(), "id": user.id, "has_key": user.has_key()},
              status=status.HTTP_200_OK,
          )
  ```

2. Cryptographic failures

- User passwords are hashed using Argon2id https://cheatsheetseries.owasp.org/cheatsheets/Password_Storage_Cheat_Sheet.html
- Using modern cryptographic algorithms (AES-GCM, PBKDF2, ECDH)
- Cryptographically safe random generator use for initialization vectors and salts

3. Injection

- Django queries use query parametrization, offering SQL injection protection
- Sanitized user input

4. Insecure design

- Secure design principles have been utilized during the whole development progress

5. Security Misconfiguration

- No unnecessary features in use
- Making sure CORS/CSRF trusted origins are correct, debug settings off

6. Vulnerable and outdated components

- Keeping components updated with the help of the CI pipeline
- Snyk, Trivy warn of outdated components

7. Identification and authentication failures

- Login/register rate limiting
- Password validation
- Invalidate/remove JWT after logout

8. Software and data integrity failures

- More of an issue for large-scale software

9. Security logging and monitoring failures

- Only logging of login attempts exist
- There should be more logging of suspicious activity

10. Server-side request forgery

- This web application does not fetch any remote resources with user-supplied URL