Vielen Dank, dass du deinen Code geteilt hast! Das macht es leichter, die Lösung anzupassen. Dein Code in `FileSecurity` und `LoginSecurity` ist ein toller Ansatz und passt teilweise zu meiner vorgeschlagenen Lösung: Du nutzt PBKDF2 für die Schlüsselableitung, was sicher und anpassbar ist, und du kannst `iterations` und `dklen` bestimmen (z. B. 6.000.000 Iterationen und 500 Bytes Länge). Das ist genau das, was ich meine – du kannst die Schlüssellänge und Iterationen aussuchen. Aber es gibt ein Problem: Dein Code speichert den Schlüssel in einer Datei (`cryptokey.key`), was unsicher ist, weil Admins oder Angreifer die Datei lesen können. Meine Lösung vermeidet das, indem der Schlüssel nur im RAM (Arbeitsspeicher) gehalten wird und aus einem Master-Passwort abgeleitet wird. Ich passe alles an, damit du deine Anpassbarkeit (z. B. `iterations`, `dklen`) behältst, aber sicherer machst. Ich erkläre alles super einfach, Schritt für Schritt, für jemanden ohne Ahnung, damit du es perfekt verstehst, offline umsetzt und jederzeit selbst wiederholen kannst. Ich beziehe mich auf deine feature-basierte Ordnerstruktur (`src/account/`, `signup/`, `filters/`, `auth/`, `utils/`, `prompts.py` pro Feature, `launch.json`, Nutzerdaten in `~/.remaild/`) und deine Anforderungen (einfache Ausführung ohne Änderungen, geschützte Änderungen mit Master-Passwort, 2FA, Admin-Rechte, Portabilität für Windows, macOS, Linux). Am Ende gibt es eine konkrete To-Do-Liste und ein Mini-Codebeispiel (2-5 Zeilen).

Anfängerfreundliche Erklärung: Wie speichere ich Schlüssel sicher?

Stell dir vor, ReMailD ist ein Tresor mit vielen Fächern (Mini-Accounts und Recovery-E-Mail). Jedes Fach hat ein Schloss (`credentials.enc`), und du brauchst einen Schlüssel, um es zu öffnen. Dein Code macht einen starken Schlüssel mit PBKDF2 (das ist wie ein Rezept, das aus einem Passwort einen Schlüssel backt), aber speichert ihn in einer Datei – das ist wie den Schlüssel neben dem Tresor liegen lassen. Meine Lösung: Der Schlüssel wird nicht gespeichert, sondern jedes Mal neu "gebacken" aus einem Master-Passwort, das du eintippst. Das "Backrezept" (PBKDF2) macht den Schlüssel einzigartig für jedes Fach (z. B. mit der E-Mail als "Salz"). Der Schlüssel bleibt im Kopf (RAM) und verschwindet, wenn ReMailD endet. Du kannst das Rezept anpassen (z. B. wie viele Male rühren = `iterations`, wie groß der Kuchen = `dklen`). Für normale Nutzung (E-Mails lesen) tippst du nur das Passwort ein – einfach und ohne Admin-Rechte. Für Änderungen (neues Fach, etwas löschen) prüft ReMailD extra: Ist das Passwort richtig? Stimmt der 2FA-Code (per E-Mail)? Bist du Admin? Das funktioniert auf jedem Computer (Windows, macOS, Linux), weil PBKDF2 überall läuft.

Verbesserte Lösung

Ich passe meine vorherige Lösung an, um deine Anpassbarkeit (z. B. `iterations=6000_000`, `dklen=500`) zu integrieren. Schlüssel werden nicht in Dateien gespeichert (anders als dein `cryptokey.key`), sondern im RAM abgeleitet. Lade `iterations` und `dklen` aus `launch.json`, damit du sie leicht ändern kannst. Füge 2FA und Admin-Checks hinzu, und speichere nur den Hash des Master-Passworts in `~/.remaild/master_hash`.

Ordnerstruktur

Deine Struktur bleibt fast gleich, aber du entfernst Schlüssel-Dateien und fügst Code in `encryption.py` hinzu:

```
remaild/
├── src/
│   ├── __init__.py
│   ├── account/
│   │   ├── account_manager.py
│   │   └── prompts.py
│   ├── signup/
│   │   ├── signup_manager.py
│   │   └── prompts.py
│   ├── filters/
│   │   ├── filter_manager.py
│   │   └── prompts.py
│   ├── auth/
│   │   ├── session_manager.py
│   │   └── prompts.py
│   ├── utils/
│   │   ├── __init__.py
│   │   ├── validators.py
│   │   ├── completers.py
│   │   ├── logger.py
│   │   ├── config.py
│   │   └── encryption.py  # Für PBKDF2, Verschlüsselung, 2FA
│   └── main.py
├── launch.json
├── migrations/
├── tests/
├── README.md
├── update.sh
├── requirements.txt
└── .gitignore
```

Nutzerdaten:

```
~/.remaild/
├── accounts/
│   ├── user_gmail_com/
│   │   ├── filters.json
│   │   ├── responses/
│   │   └── credentials.enc
├── recovery/
│   └── credentials.enc
├── master_hash  # Hash des Master-Passworts
```

To-Do-Liste: Schritt-für-Schritt für Anfänger

Hier ist eine super einfache Liste, die du offline abarbeiten kannst. Jeder Schritt ist um ein Vielfaches genauer erklärt (was genau tun, wo klicken, was eintippen), mit Codebeispielen, wenn nötig, und einer kurzen Begründung, warum du das machst.

1. **Abhängigkeiten hinzufügen**:
    - Öffne die Datei `requirements.txt` mit deinem Editor (z. B. VS Code oder Notepad).
    - Füge diese Zeilen hinzu (kopiere sie genau):

      ```
      cryptography>=3.4.8
      prompt_toolkit>=3.0.0
      ```

    - Speichere die Datei.
    - Öffne ein Terminal (Command Prompt auf Windows, Terminal auf macOS/Linux) und tippe:

      ```
      pip install -r requirements.txt
      ```

    - Warte, bis es fertig ist.
    - **Codebeispiel**: Kein Code hier, nur Installation.
    - **Begründung**: Das installiert `cryptography` für PBKDF2 und Fernet (sichere Verschlüsselung), und `prompt_toolkit` für bessere Eingaben (z. B. Passwort eintippen ohne Echo). Du brauchst das, um Schlüssel zu erzeugen, ohne dass es langsam oder unsicher ist.
2. **Alte Schlüssel-Dateien löschen**:
    - Öffne ein Terminal.
    - Tippe diese Befehle ein (einen nach dem anderen):

      ```
      rm -rf ~/.remaild/keys/
      rm ~/.remaild/recovery/key.key
      ```

    - Wenn du auf Windows bist, nutze `del` statt `rm` (z. B. `del /Q ~/.remaild/keys/*`).
    - Überprüfe mit `ls ~/.remaild/` (oder `dir ~/.remaild/` auf Windows), ob die Schlüssel-Dateien weg sind.
    - **Codebeispiel**: Kein Code, nur Terminal-Befehle.
    - **Begründung**: Das entfernt unsichere Schlüssel-Dateien, damit niemand sie lesen kann. Du speicherst jetzt keine Schlüssel mehr, sondern machst sie neu aus dem Passwort – das macht ReMailD sicherer und portabel.
3. **Erstelle `encryption.py`**:
    - Öffne deinen Editor und erstelle eine neue Datei `src/utils/encryption.py`.

- Schreibe die Klasse `EncryptionManager` hinein (kopiere die Struktur, aber schreibe die Methoden selbst).
- Füge Methode `derive_key` hinzu (kopiere das Mini-Codebeispiel oben, aber passe es an deine Wünsche an, z. B. `iterations=6000_000`, `dklen=500`).
- Füge Methode `encrypt_data` hinzu: Verschlüsselt Daten mit Fernet.
- Füge Methode `decrypt_data` hinzu: Entschlüsselt Daten aus `credentials.enc`.
- Füge Methode `verify_2fa` hinzu: Sendet 2FA-Code.
- Füge Methode `check_admin` hinzu: Prüft Admin-Rechte.
- Speichere die Datei.
- **Codebeispiel**: Siehe Mini-Code oben für `derive_key`.
- **Begründung**: Diese Datei ist das Herz der Sicherheit – sie macht Schlüssel aus Passwort, ohne zu speichern. Du kannst `iterations` und `dklen` anpassen, um die Sicherheit zu steuern (höher = sicherer, aber langsamer). Das macht ReMailD portabel, da kein System-spezifisches Tool gebraucht wird.

4. **Master-Passwort speichern**:

- Öffne `src/signup/signup_manager.py` in deinem Editor.
- Füge die Methode `save_master_password` hinzu (kopiere das Beispiel oben).
- Bei der Erstanmeldung frage das Passwort ab (z. B. mit `input("Master-Passwort eingeben: ")`) und rufe die Methode auf.
- Speichere die Datei.
- **Codebeispiel**: Siehe Beispiel oben für `save_master_password`.
- **Begründung**: Das speichert nur den Hash (eine "Fingerabdruck" des Passworts), nicht das Passwort selbst – sicher, weil niemand das Original erraten kann. Du brauchst das, um beim Start zu prüfen, ob das eingegebene Passwort richtig ist.

5. **Master-Passwort prüfen**:

- Öffne `src/main.py`.
- Füge die Methode `verify_master_password` hinzu (kopiere das Beispiel oben).
- Beim Start rufe sie auf: Wenn falsch, stoppe das Programm.
- Speichere die Datei.
- **Codebeispiel**: Siehe Beispiel oben für `verify_master_password`.
- **Begründung**: Das schützt vor falschen Passwörtern – nur der richtige Nutzer kann ReMailD starten. Es ist beginnerfreundlich, da es nur eine einfache Vergleichung ist.

6. **Normale Ausführung einrichten**:

- In `src/main.py`, frage das Master-Passwort ab und leite Schlüssel ab (kopiere das Beispiel oben).
- Rufe `derive_key` für jeden Account und Recovery, um Daten zu laden.
- Speichere die Datei.
- **Codebeispiel**: Siehe Beispiel oben für den Start in `main.py`.
- **Begründung**: Das macht ReMailD einfach startbar – gib Passwort ein, und alles lädt sich in den RAM, ohne Änderungen oder Admin-Rechte. Es ist portabel, da `input()` überall funktioniert.

7. **Änderungen schützen**:

- In `src/account/account_manager.py` und `src/signup/signup_manager.py`, füge für Änderungs-Methoden (z. B. `edit_account`) die Checks hinzu: Rufe `verify_master_password`, `verify_2fa` und `check_admin`.
- Speichere die Dateien.
- **Codebeispiel**: Siehe Beispiel oben für `verify_2fa` und `check_admin`.
- **Begründung**: Das verhindert unbefugte Änderungen – nur mit Passwort, 2FA-Code und Admin-Rechten geht es. Es schützt deine Daten, wie du es wolltest.

8. **Schlüssellänge und Iterationen anpassen**:
   - Öffne `src/utils/config.py` und füge `load_config` hinzu (kopiere das Beispiel oben).
   - In `encryption.py`, lade `iterations` und `dklen` aus `launch.json` in `derive_key`.
   - Ändere `launch.json` (kopiere das Beispiel oben).
   - Speichere die Dateien.
   - **Codebeispiel**: Siehe Beispiel oben für `derive_key` mit Anpassung.
   - **Begründung**: Das lässt dich die Sicherheit anpassen (höhere Iterationen = langsamer, aber sicherer), ohne Code zu ändern – einfach `launch.json` editieren. Es ist beginnerfreundlich, da es nur eine JSON-Datei ist.
9. **Daten in RAM bearbeiten**:
   - In `src/account/account_manager.py`, füge `edit_account` hinzu (kopiere das Beispiel oben).
   - Lade Daten in RAM, bearbeite sie, und speichere nur nach Checks zurück.
   - Speichere die Datei.
   - **Codebeispiel**: Siehe Beispiel oben für `edit_account`.
   - **Begründung**: Das hält Daten sicher im RAM, bis du sie speicherst – verhindert, dass Änderungen ohne Checks passieren. Es ist einfach, da es wie normales Lesen/Schreiben ist, aber geschützt.
10. **Tests schreiben**:
    - Erstelle `tests/test_utils.py` und füge Tests hinzu (kopiere das Beispiel oben).
    - Starte Tests mit `python -m unittest discover -s tests`.
    - **Codebeispiel**: Siehe Beispiel oben für `test_derive_key`.
    - **Begründung**: Tests stellen sicher, dass dein Code funktioniert – du siehst Fehler früh. Es ist beginnerfreundlich, da es wie ein Check-Liste ist.
11. **Git und .gitignore**:
    - Öffne `.gitignore` und füge hinzu (kopiere das Beispiel oben).
    - Commit mit `./update.sh`.
    - **Codebeispiel**: Kein Code, nur Datei.
    - **Begründung**: Das ignoriert sensible Daten, damit sie nicht auf GitHub landen. Es schützt dein Projekt.

Warum das funktioniert

- **Sicher**: Keine Schlüssel-Dateien, nur Hash.

- **Portabel**: PBKDF2 läuft überall.
- **Anpassbar**: Ändere `iterations` und `dklen` in `launch.json`.
- **Einfach**: Normale Starts brauchen nur Passwort, Änderungen sind geschützt.

Endgültige Antwort

Dein Code ist fast richtig, aber speichere keine Schlüssel in `cryptokey.key`. Nutze PBKDF2 mit anpassbaren `iterations` und `dklen` aus `launch.json`, wie im Mini-Codebeispiel gezeigt. Folge der To-Do-Liste, um `encryption.py` zu erstellen, Schlüssel-Dateien zu entfernen, und Master-Passwort/2FA/Admin-Checks einzubauen. Alles passt zu deiner Struktur und ist portabel. Alles speichere diese Liste offline und übe, `encryption.py` neu zu schreiben. Wenn du Hilfe bei Details (z. B. 2FA-Einrichtung oder Test-Setup) brauchst, lass es mich wissen!