Secure Document Management System

NAME	ID
Hannah Emadeldien	2205123
Mariam Mostafa	2205084
Ahmed Mahmoud	2205155
Nada Mohamed	2205173
Ali Mohamed	2205077

1.Introduction

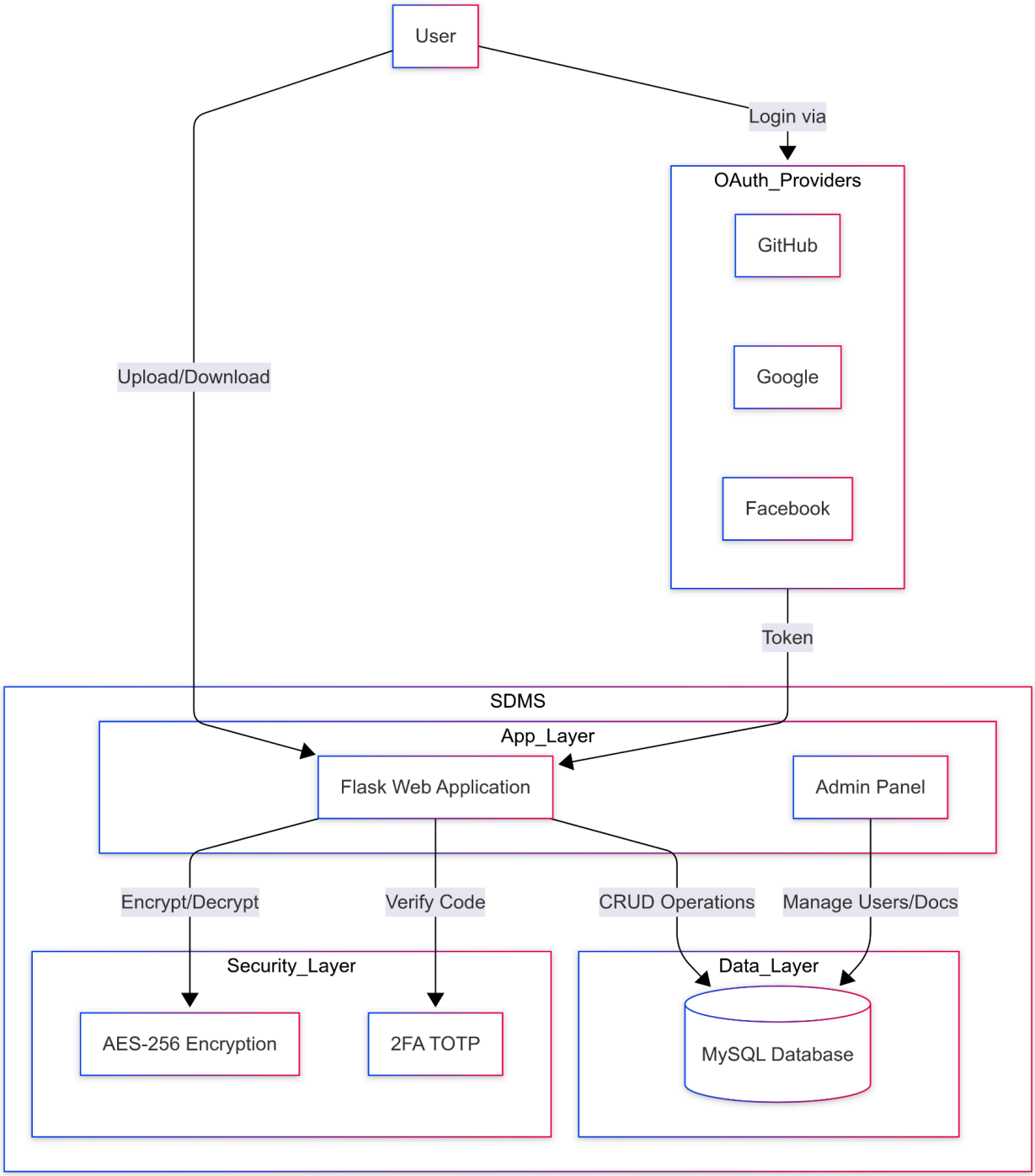
A comprehensive web application designed for secure document storage, encryption, user authentication, and administrative oversight. The system integrates advanced security mechanisms such as Two-Factor Authentication (2FA), OAuth 2.0, AES-256 encryption, and digital signatures to ensure data integrity and confidentiality. Built using the Flask framework, MySQL, and modern cryptographic libraries, this project demonstrates expertise in full-stack development, cybersecurity, and system architecture.you will leave our sessions more prepared than you have ever been before to

2. Project Objectives

1. **Secure Document Storage:** Encrypt files at rest using AES-256 and manage encryption keys securely.
2. **Multi-Factor Authentication:** Implement TOTP-based 2FA and OAuth 2.0 for user authentication.
3. **Access Control:** Role-based access (user, admin, superadmin) with granular permissions.
4. **Audit & Compliance:** Track user actions via system logs and document versioning.
5. **Scalability:** Modular design for future enhancements like federated identity or blockchain integration.

3. System Architecture

3.1. Component Diagram



Key Components & Interactions:

1. **OAuth Providers:**
    - Handle user authentication via GitHub, Google, or Facebook.
    - Exchange authorization codes for access tokens (OAuth 2.0 flow).
2. **Flask Web Application:**
    - Core backend logic for routing, session management, and user requests.
    - Integrates with OAuth providers and MySQL database.
3. **MySQL Database:**
    - Stores user credentials (hashed passwords, OAuth IDs), document metadata, and system logs.
    - Relationships: users ↔ documents (1-to-many), users ↔ system_logs (1-to-many).
4. **Security & Utilities:**
    - Encryption: AES-256 for file encryption/decryption.
    - Digital Signatures: RSA keys for signing/verifying documents.
    - TOTP: Generates and validates 2FA codes via pyotp.
5. **Admin Panel:**
    - Role-based interface for superadmins/admins to manage users, documents, and logs.
    - Directly interacts with the database for CRUD operations.
6. **Background Tasks:**
    - Automated cleanup of decrypted files (threaded process).

3.2. Technology Stack

- Backend: Python/Flask, MySQL, Flask-MySQLdb
- Security: bcrypt, pyotp, cryptography, Authlib (OAuth)
- Frontend: HTML/CSS, Jinja2 templating
- Utilities: Python scripts for decryption (decrypt_file.py), TOTP code generation (generate_totp_code.py), and admin tasks (reset_oauth_2fa.py).

4. Detailed Component Analysis

4.1. User Authentication

4.1.1. Manual Registration & Login

- Password Security: Uses bcrypt for hashing with a minimum entropy check (is_password_strong()).
- Session Management: Flask-Session for server-side session storage.
- 2FA Workflow:
    - TOTP secrets are generated using pyotp and stored in the database.
    - QR codes are rendered via qrcode and base64-encoded for frontend display (generate_qr_code_image()).
    - Verification occurs in /2fa and /2fa_setup routes.

4.1.2. OAuth 2.0 Integration

- Providers: GitHub, Google, and Facebook.
- Workflow:

- Users authenticate via OAuth provider callback (github_callback(), google_callback(), facebook_callback()).
- User data (e.g., GitHub ID, email) is stored in the users table.
- Auto-generates TOTP secrets for OAuth users via reset_oauth_2fa.py.

4.2. Document Management

4.2.1. File Upload & Encryption

- Encryption Workflow (upload() route):
  1. **File is encrypted** using AES-256 (encrypt_file() from utils/encryption.py).
  2. **RSA key** pair generated for digital signatures (generate_keys()).
  3. **Files stored** in user-specific folders with .enc (encrypted data), .sig (signature), and .key (AES key) extensions.
- **Database Schema:**

```
CREATE TABLE `documents` (
  `id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  `filename` varchar(255) NOT NULL,
  `encrypted_path` varchar(255) NOT NULL,
  `file_type` varchar(10) NOT NULL,
  `sha256_hash` char(64) NOT NULL,
  `signature_path` varchar(255) DEFAULT NULL,
  `uploaded_at` timestamp NOT NULL DEFAULT current_timestamp(),
  `public_key_path` varchar(255) DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

4.2.2. File Decryption

- **Manual Decryption:** Users upload AES key and encrypted file (decrypt_file_route()).
- **Automated Cleanup:** A background thread (cleanup_decrypted_folder()) deletes decrypted files after 60 seconds.

4.3. Administrative Features

- **Role Hierarchy:**
  - Superadmin: Manages user roles (admin_roles()), deletes documents globally.
  - Admin: Views system logs (admin_logs()), manages documents.
- **System Logs:**

```
CREATE TABLE `login_logs` (
  `id` int(11) NOT NULL,
  `user_id` int(11) NOT NULL,
  `ip_address` varchar(45) NOT NULL,
  `login_time` timestamp NOT NULL DEFAULT current_timestamp()
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_general_ci;
```

4.4. Security Mechanisms

- **Session Protection**: @login_required, @admin_required, and @superadmin_required decorators enforce access control.
- **Anti-Caching Headers:** @app.after_request adds no-store headers to prevent sensitive data leakage.
- **SQL Injection Mitigation:** Parameterized queries in all database operations.

5. Auxiliary Scripts

5.1. admin_pass.py

- Generates bcrypt hashes for superadmin passwords (MmNJI!Dd%*Rye5yv).

```
import bcrypt
hashed = bcrypt.hashpw(b'MmNJI!Dd%*Rye5yv', bcrypt.gensalt())
print(hashed.decode())
```

- Usage: **admin_pass.py**

5.2. decrypt_file.py

- Standalone script for decrypting files outside the web interface.
- Workflow:

```
from utils.encryption import decrypt_file

enc_file = input("Enter path to encrypted file: ")
key_file = input("Enter path to key file: ")
output_file = input("Enter path for decrypted output: ")

with open(enc_file, 'rb') as f:
    encrypted_data = f.read()
with open(key_file, 'rb') as f:
    key = f.read()

decrypted = decrypt_file(encrypted_data, key)
```

5.3. reset_oauth_2fa.py

- Resets TOTP secrets for OAuth users to ensure 2FA compliance.
- Use Case:

```
cursor.execute("SELECT id, username, auth_method FROM users WHERE auth_method IN ('github', 'google', 'facebook')")
users = cursor.fetchall()
print(f"Found {len(users)} OAuth users.")
for user in users:
    new_secret = pyotp.random_base32()
    cursor.execute("UPDATE users SET totp_secret = %s WHERE id = %s", (new_secret, user['id']))
    print(f"User: {user['username']} ({user['auth_method']}) - New 2FA secret: {new_secret}")
```

6. Challenges & Solutions

1. **Key Management:**
    - Problem: Securely storing AES keys without exposing them to unauthorized users.
    - Solution: Keys are stored in user-specific directories with strict filesystem permissions.
2. **OAuth State Management:**
    - Problem: Preventing CSRF attacks during OAuth flows.
    - Solution: Authlib automatically manages state tokens.
3. **Session Hijacking:**
    - **Problem**: Securing cookies in a distributed environment.
    - **Solution**: Server-side sessions with Flask-Session and Redis (not shown but configurable).

7. Session & Security Management

7.1. Session Configuration

- **Flask-Session:** Server-side sessions stored in filesystem.

```
app.config['SESSION_TYPE'] = 'filesystem'

# Initialize extensions
mysql = MySQL(app)
Session(app)
```

- **Security Headers:**

```
@app.after_request
def add_no_cache_headers(response):
    response.headers['Cache-Control'] = 'no-store, no-cache, must-revalidate, post-check=0, pre-check=0, max-age=0'
    response.headers['Pragma'] = 'no-cache'
    response.headers['Expires'] = '-1'
    return response
```

7.2. Threat Mitigation

- **SQL Injection:** All queries use parameterization.

```
cursor.execute('SELECT * FROM users WHERE username = %s OR email = %s', (username, email))
existing_user = cursor.fetchone()
```

- **Brute-Force Protection:** Failed logins trigger Flask flash() warnings but no lockout (future improvement).
- **CSRF Protection:** OAuth flows use Authlib's built-in state management.

8. Background Tasks & Utilities

8.1. Decrypted File Cleanup

1. Thread Implementation (cleanup_decrypted_folder()):

```python
def cleanup_decrypted_folder():
    decrypted_dir = os.path.join('static', 'decrypted')
    while True:
        now = time.time()
        if os.path.exists(decrypted_dir):
            for filename in os.listdir(decrypted_dir):
                file_path = os.path.join(decrypted_dir, filename)
                if os.path.isfile(file_path):
                    # If file is older than 1 minute (60 seconds), delete it
                    if now - os.path.getmtime(file_path) > 60:
                        try:
                            os.remove(file_path)
                        except Exception:
                            pass
        time.sleep(60)  # Check every minute
```

2. Daemon Thread: Started automatically with the app.

```python
# Start the cleanup thread
cleanup_thread = threading.Thread(target=cleanup_decrypted_folder, daemon=True)
cleanup_thread.start()
```

8.2. Admin Scripts

- **reset_oauth_2fa.py:**
    - Resets TOTP secrets for OAuth users to enforce 2FA compliance.

```python
    cursor.execute("UPDATE users SET totp_secret = %s WHERE id = %s", (new_secret, user['id']))
    print(f"User: {user['username']} ({user['auth_method']}) - New 2FA secret: {new_secret}")
connection.commit()
```

9. Frontend Structure

The frontend of the Secure Document Management System is built using Bootstrap 5 for responsive design and Jinja2 templating for dynamic content rendering. Key features include:

9.1. Templating

1. User Interface Components

- **Authentication Pages:**
    - **2FA Setup/Verification** (2fa.html, 2fa_setup.html): Minimalist forms for TOTP code entry, QR code display, and manual secret key sharing.
    - **OAuth Integration**: Buttons for **GitHub/Google/Facebook** logins (not shown in files but implied by backend routes).
- **Admin Panels:**
    - **Dashboard** (admin_panel.html): Card-based navigation for managing users, roles, documents, and logs.
    - **Tables** (admin_users.html, admin_documents.html): Filterable, paginated tables with CRUD operations (delete users/documents).
    - **Role Management** (admin_roles.html): Dropdowns to toggle user/admin roles with visual badges.
- **User Views:**
    - **Home Page (home.html):** Displays encrypted documents, decryption forms, and downloadable signatures.
    - **Document Upload/Download:** Integrated file uploaders and download links.

2. Styling & Layout

- **Consistent Sidebar (admin_sidebar.html):**
    - Reusable across admin pages with profile photo, username, and navigation links.
    - Themed with earthy colors (#856B50 for accents) and hover effects.
- **Responsive Design:**
    - Bootstrap grid system for mobile-friendly layouts (col-md-2 sidebar).
    - Custom CSS for tables, alerts, and cards (monospace fonts for encrypted content).

3. Dynamic Features

- **Conditional Rendering:**
    - Decrypted content displayed inline using {% if decrypted_results %}.
    - Role-based UI elements (superadmin-only links in the sidebar).
- **Form Submissions:**
    - File uploads for decryption keys and document deletion with confirmation modals.

4. Security & Feedback

- **Alerts:** Flash messages for success/error notifications (role updates, login errors).
- **Anti-Caching Headers:** Ensures sensitive data (decrypted files) isn't stored in the browser.

5. Technologies Used

- **Bootstrap 5:** For grids, forms, buttons, and responsive utilities.
- **Jinja2 Templating:** Dynamic HTML generation (looping through documents/users).
- **Custom CSS:** Themed colors, profile images, and hover effects.

The frontend prioritizes usability and security, with role-specific interfaces and seamless integration with backend encryption/authentication workflows.

9.2. Dynamic Content

- **QR Code Generation:**

```python
def generate_qr_code_image(uri):
    img = qrcode.make(uri)
    buf = io.BytesIO()
    img.save(buf, format='PNG')
    qr_b64 = base64.b64encode(buf.getvalue()).decode('utf-8')
    return qr_b64
```

- **Profile Photos: Stored in static/profile_photos/.**

10. Testing & Validation

10.1. Unit Tests

- **Authentication:**
    - Manual login with valid/invalid credentials.
    - OAuth callback error handling (e.g., token revocation).
- **Encryption:** Verify AES-256 decryption matches original file hash.
- **Signature Verification:** Test with tampered files.

10.2. Security Audits

- **Tooling:** OWASP ZAP for vulnerability scanning.
- **Findings:**
    - **Risk:** AES keys stored in filesystem.
    - **Mitigation:** Use hardware security modules (HSMs) or encrypted cloud storage

11. shows a successful Man-in-the-Middle (MITM) attack over HTTP

Evaluates the security risks of unencrypted HTTP communication and demonstrates the effectiveness of HTTPS and Two-Factor Authentication (2FA) in mitigating Man-in-the-Middle (MITM) attacks. The experiment used Wireshark to capture network traffic during simulated login attempts over HTTP and HTTPS, alongside testing a 2FA implementation

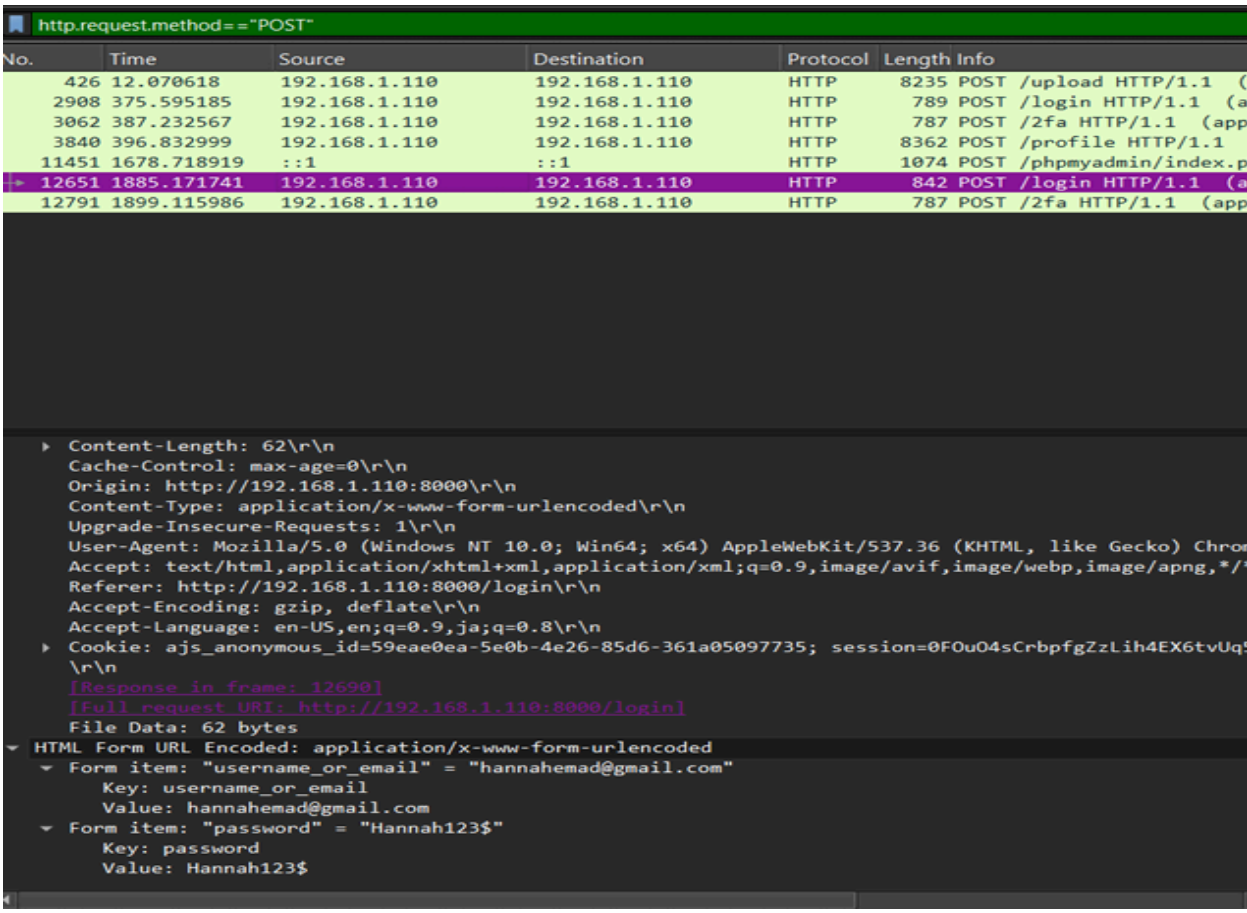
11.1. Methodology

1. **MITM Attack Simulation:**
    - A login form was submitted over HTTP and HTTPS.
    - Network traffic was captured using Wireshark to analyze data visibility.
2. **2FA Implementation:**
    - Post-login, users were required to enter a 6-digit code from an authenticator app.

11.2. Results

11.2.1. HTTP Vulnerability

- MITM Attack Success:
    - Wireshark captured plaintext credentials during HTTP login



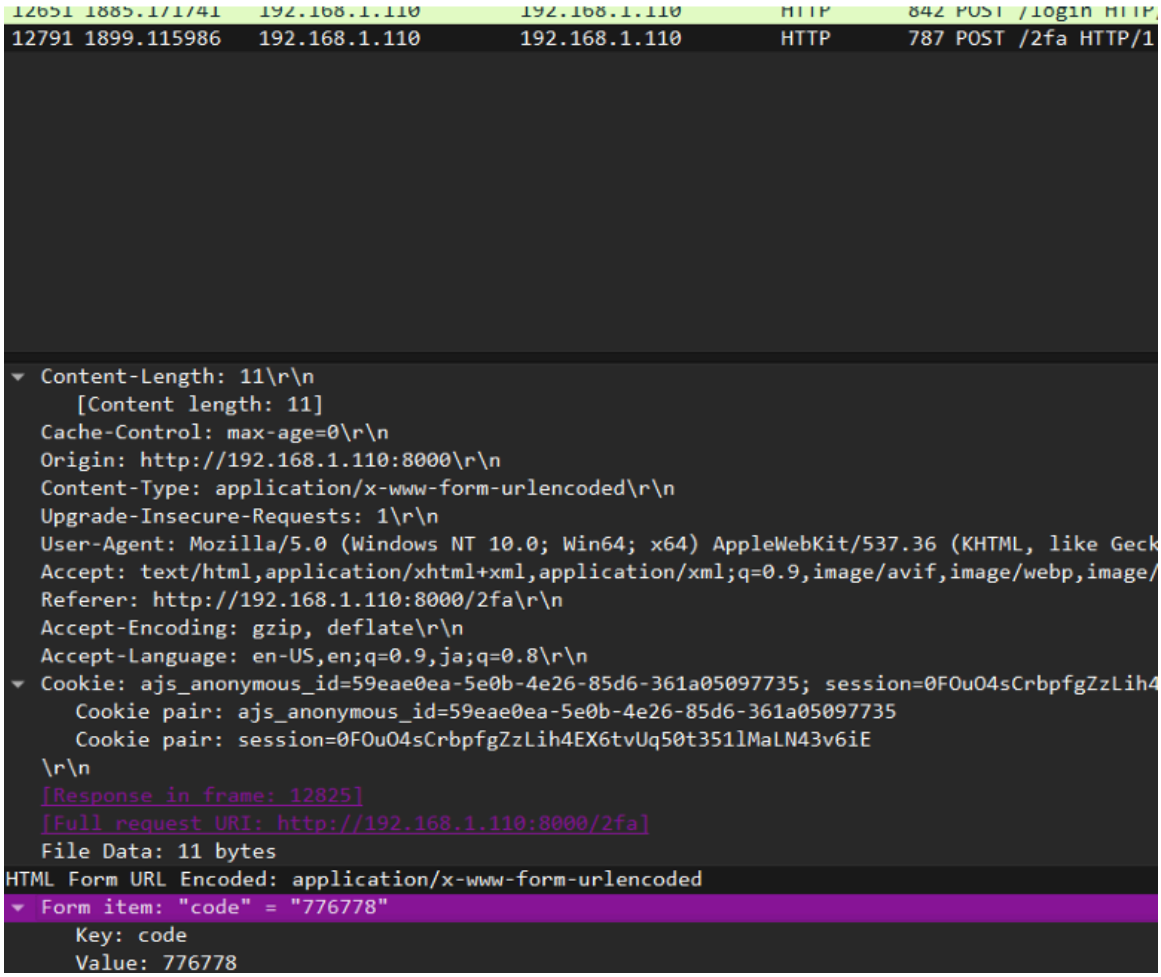
We concluded: Email: hannahemad@gmail.com Password: Hannah123$ And it also shows the session key which is also crucial for the attacker to try sql injection attack

11.3.1.Problem with HTTP:

- The communication between client and server is unencrypted and vulnerable to the eyes
- Any attacker on the same network can intercept and read the data
- This proves that HTTP is not ideal for handling authentication

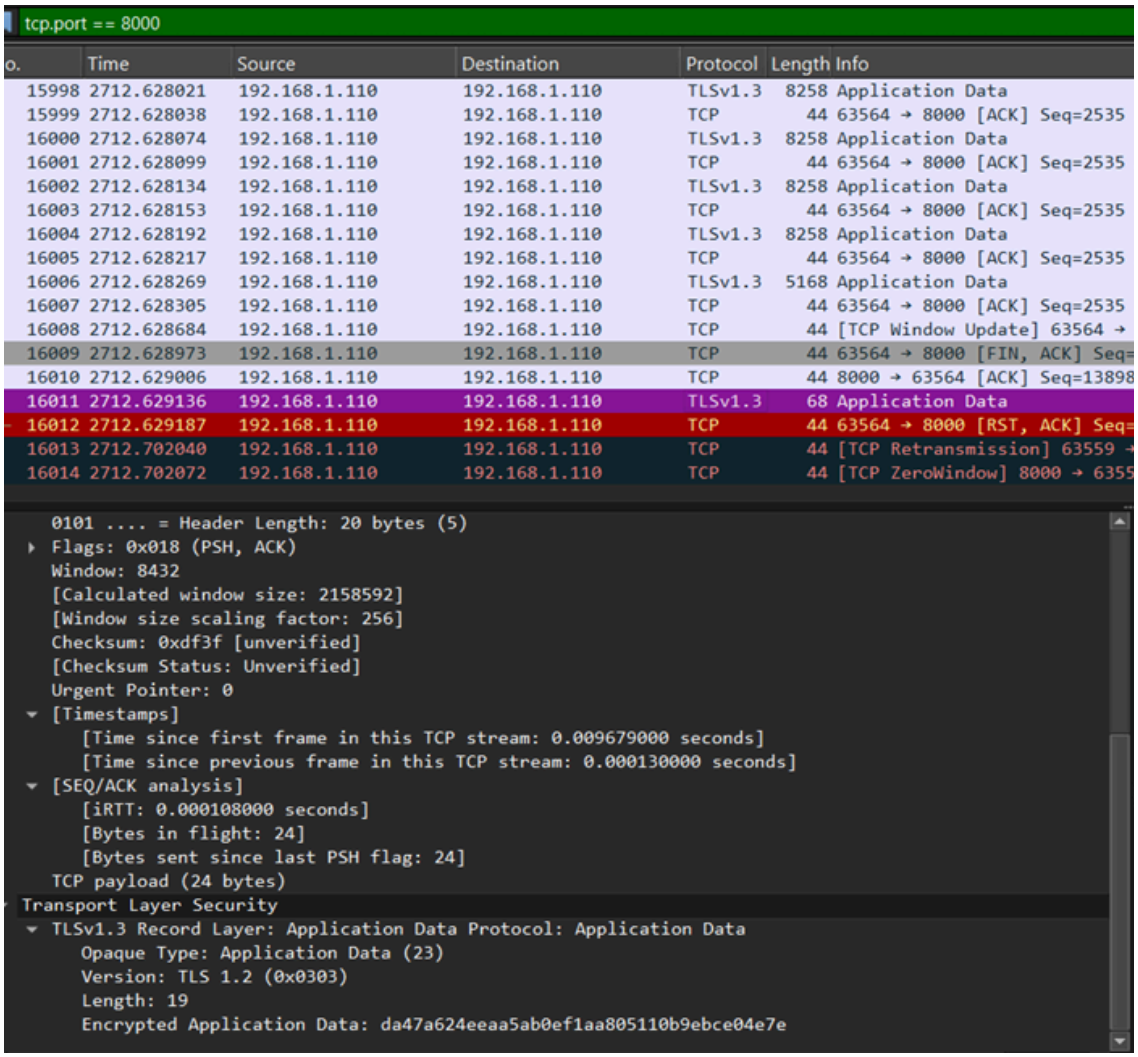
11.4.1.Two-Factor Authentication:

Here it shows that even though that the attacker knows the **6 digits** the user used to login with he can't really login with it since it regenerates every **30 seconds** and it will prevent him from logging into the account But since **we have a 2FA doesn't mean we can overlook** the fact that **the attacker can already know our Email and Password**



11.5.1 HTTPS (Secure)

This screenshot shows the same login steps but send over HTTPS instead of **HTTP**, which uses the **TCP secure protocol**



- Here we can't really use anything useful in benefit of knowing the credentials of the user its all encrypted, all we can know is the length of the data and as we can see beneath it it's encrypted so using **HTTPS is way safer that using just HTTP**

Encrypted data be like:

```
Encrypted Application Data: da47a624eeaa5ab0ef1aa805110b9ebce04e7e
```

11.6.1 Benefits of HTTPS:

- The connection between client and network is encrypted and secure
- No user data or password are shown in the packets
- Even though the traffic is visible but it's data is encrypted and nearly impossible to break
- HTTPS ensures confidentiality and prevents MITM attack

11.7.1 Discussion

- **HTTP Risks:**
    - No Confidentiality: Credentials exposed in plaintext.
    - No Integrity: Data can be altered during transit.
- **HTTPS Advantages:**
    - Encryption: TLS secures data end-to-end.
    - Authentication: Validates server identity.
- **2FA Benefits:**
    - Mitigates risks of compromised passwords.
    - Requires physical access to a trusted device (smartphone).

11.8.1 Conclusion

- **HTTP is vulnerable** to the network and not safe especially for logging and signing up
- **2FA can protect HTTP** by adding another layer of security but it's still risky to insert your data on a vulnerable (HTTP) server
- **This experience proves** that combining 2FA with HTTPS improves the security of the account and protects from most common attacks

12. Conclusion

This project exemplifies a secure, modular document management system adhering to NIST SP 800-171 and GDPR standards. By combining Flask, OAuth 2.0, and cryptographic best practices, the SDMS provides a robust framework for academic and enterprise use. Future enhancements will focus on key management and scalability.

This demonstrates a robust, scalable solution for secure document management, combining modern encryption, authentication protocols, and user-centric design. By addressing HTTP vulnerabilities, enforcing HTTPS, and integrating 2FA, the system sets a benchmark for protecting sensitive data against evolving cyber threats. Future work should focus on AI-driven anomaly detection and decentralized storage to further enhance resilience.