Università degli Studi di Messina

Department of MIFT

Multi-Factor Authentication System

Web-Based Application for Secure Authentication

Project Report

Student Name:	Shafiqullah Nasiree
Student ID:	550284
Supervisors:	Prof. Massimo Villari
	Prof. Armando Ruggeri
Academic Year:	2024-2025
Submission Date:	January 2026

Abstract

This report presents the design, implementation, and comprehensive security analysis of a web-based Multi-Factor Authentication (MFA) system developed to demonstrate fundamental principles of secure identity verification in distributed computing environments. The implemented system integrates password-based authentication with temporally constrained One-Time Passwords (OTPs) delivered through an independent communication channel—specifically, the Telegram messaging platform—thereby establishing a defense-in-depth security architecture.

The system architecture incorporates five sequential security validation mechanisms: cryptographically secure random number generation utilizing Python's `secrets` module, temporal validity enforcement with 60-second expiration windows, attempt-based brute-force mitigation limiting verification to three attempts, IP address binding for session integrity verification, and replay attack prevention through single-use enforcement protocols. These mechanisms collectively address prevalent authentication vulnerabilities including credential theft, session hijacking, and man-in-the-middle attacks.

Implementation was realized using Python 3.11 with the Flask web framework for backend API development, SQLite for persistent database storage with automatic schema initialization, werkzeug.security for password hashing, standard HTML5/CSS3/JavaScript for client-side interface construction, and the Telegram Bot API for out-of-band communication. The database provides ACID-compliant persistent storage for user credentials (stored as hashed passwords) and OTP lifecycle management, eliminating data volatility across server restarts.

Comprehensive testing validated successful mitigation of common attack vectors including credential replay, brute-force enumeration, and remote session hijacking attempts. The system demonstrates platform independence through Docker containerization, ensuring deployment portability across heterogeneous infrastructure including ARM-based embedded devices. This work contributes to practical understanding of authentication security mechanisms, demonstrating how multiple independent verification factors substantially elevate the difficulty threshold for unauthorized access beyond single-factor authentication paradigms.

Keywords: Multi-Factor Authentication, One-Time Password, Out-of-Band Verification, Session Security, Cryptographic Security, Flask Framework, SQLite Database, Password Hashing, Docker Containerization, Attack Mitigation

Contents

List of Figures

1 Introduction

1.1 Context and Motivation

Authentication mechanisms constitute the foundational security control upon which access control systems fundamentally depend. In contemporary computing environments, the traditional paradigm of username-password authentication has demonstrated significant vulnerabilities when deployed as the sole verification mechanism. Large-scale credential breaches, sophisticated phishing campaigns, and the widespread prevalence of weak or reused passwords collectively undermine the security assurances provided by knowledge-based authentication factors operating in isolation.

The fundamental weakness inherent in single-factor authentication stems from its reliance on a single point of failure. Should an attacker successfully obtain or derive the authentication credential through any of numerous attack vectors—including but not limited to keylogging, social engineering, database compromise, or brute-force enumeration—complete authentication bypass results. This vulnerability has been extensively documented in security incident reports, with compromised credentials identified as the primary attack vector in a substantial proportion of unauthorized access incidents.

Multi-Factor Authentication addresses this fundamental limitation by requiring the presentation of credentials from multiple independent factor categories. The security premise underlying MFA rests on the assumption that compromising multiple independent factors presents substantially greater operational difficulty than compromising any single factor in isolation. This defense-in-depth approach aligns with established security engineering principles and represents current best practice for identity verification in systems requiring elevated security assurances.

1.2 Project Objectives and Academic Scope

This project was undertaken to provide comprehensive practical experience in the design, implementation, and analysis of a complete authentication system incorporating multiple security mechanisms. The work bridges theoretical security concepts with concrete implementation challenges, demonstrating how abstract security principles manifest in functional software systems. Specific learning objectives include:

- Understanding authentication factor independence and its security implications
- Practical application of cryptographic primitives in security-critical contexts
- Implementation of temporal constraints in authentication workflows
- Design and implementation of attack mitigation mechanisms

- Analysis of threat models and security trade-offs
- Deployment considerations for containerized security-critical applications
- Database integration for persistent storage and password security

The project scope was deliberately constrained to maintain pedagogical focus on fundamental security principles rather than production infrastructure concerns. This approach permits concentrated examination of core authentication concepts without the complexity overhead associated with enterprise-grade deployment requirements such as distributed database architectures, comprehensive audit logging infrastructure, or advanced session management frameworks.

1.3 System Overview

The implemented system provides two-factor authentication combining:

1. **Factor 1 (Knowledge):** Username and password credentials (passwords stored as hashes)
2. **Factor 2 (Possession):** Physical access to a Telegram account receiving time-limited OTPs

The authentication workflow proceeds through discrete phases: initial credential verification against SQLite database, cryptographically secure OTP generation, database insertion with expiration timestamps, out-of-band delivery via Telegram, and multi-criteria OTP validation with database-backed security checks. The system incorporates five distinct security checks during OTP verification, each addressing specific attack vectors. OTPs are constrained to a 60-second validity window and support a maximum of three verification attempts, after which the credential is invalidated and complete re-authentication is mandated.

2 Background and Theoretical Foundations

2.1 Authentication and Authorization Distinctions

Authentication and authorization represent distinct but complementary security functions that are frequently conflated in informal discourse. **Authentication** constitutes the process of verifying the claimed identity of an entity attempting to access a system. **Authorization**, conversely, determines which resources or operations the authenticated entity is permitted to access. This work focuses exclusively on authentication mechanisms, specifically multi-factor identity verification.

The security of any authorization system fundamentally depends on the integrity of the underlying authentication mechanism. Ineffective authentication directly compromises authorization controls, as unauthorized entities can impersonate legitimate users and inherit their access privileges. This dependency relationship establishes authentication as a critical security primitive requiring rigorous design and implementation.

2.2 Authentication Factor Categories

Authentication mechanisms traditionally employ credentials from one or more of the following factor categories:

- **Knowledge Factors (Something You Know):** Information possessed exclusively by the legitimate user, including passwords, personal identification numbers (PINs), security questions, or passphrases. Knowledge factors are susceptible to theft through observation, social engineering, or brute-force enumeration.
- **Possession Factors (Something You Have):** Physical or digital artifacts uniquely possessed by the legitimate user, including hardware tokens, smart cards, mobile devices, or cryptographic keys. Possession factors require physical theft or sophisticated duplication attacks for compromise.
- **Inherence Factors (Something You Are):** Biometric characteristics unique to the user, including fingerprints, facial geometry, iris patterns, or voice characteristics. Inherence factors resist theft but raise privacy concerns and may be subject to spoofing attacks.

Effective multi-factor authentication requires credentials from at least two distinct categories. The implemented system employs knowledge factors (password) and possession factors (Telegram access on mobile device), providing genuine factor independence.

2.3 One-Time Password Mechanisms

One-Time Passwords represent authentication credentials valid for a single session or transaction. Unlike static passwords, which remain valid until explicitly changed, OTPs automatically expire after use or after a defined time interval. This temporal limitation substantially reduces the attack window for credential interception.

Two primary OTP generation methodologies exist:

1. **HMAC-Based OTP (HOTP):** Generates authentication codes based on a counter value and a shared secret key using HMAC (Hash-based Message Authentication Code). The counter increments after each successful authentication.

2. **Time-Based OTP (TOTP):** Generates authentication codes based on the current timestamp and a shared secret key, typically producing new codes at fixed intervals (commonly 30 seconds).

The implemented system employs a hybrid approach: OTPs are randomly generated using cryptographically secure pseudo-random number generators (CSPRNGs) upon successful password verification, stored in SQLite database with timestamp metadata, and validated against a fixed expiration interval. This approach provides the security benefits of time-limitation without requiring clock synchronization between client and server or the complexity of shared secret management inherent in TOTP implementations.

2.4 Out-of-Band Authentication

Out-of-band (OOB) communication refers to the transmission of authentication data through a channel separate from the primary authentication interface. In the implemented system, while users authenticate through a web browser interface, OTP delivery occurs via the Telegram messaging platform—a completely independent communication channel operating on distinct infrastructure.

The security advantages of OOB communication include:

- **Channel Separation:** Compromise of the web session does not automatically grant access to the OTP delivery channel, requiring attackers to simultaneously compromise two independent communication paths.
- **Physical Device Requirement:** The attacker must possess or have remote access to the physical device receiving OTPs, substantially increasing operational complexity beyond purely digital attacks.
- **Alert Mechanism:** Legitimate users receive OTP notifications even during unauthorized access attempts, providing an implicit intrusion detection capability.
- **Independence from Web Security:** OTP delivery does not depend on the security of TLS/HTTPS, browser state, or web application vulnerabilities, as it operates through entirely separate infrastructure.

Telegram was selected as the OOB channel for this implementation due to its widespread availability, well-documented API, instant message delivery, infrastructure reliability, and absence of per-message costs compared to SMS-based alternatives.

2.5 Common Authentication Threat Vectors

The implemented security mechanisms address several well-documented authentication attack vectors:

- **Credential Theft via Phishing:** Attackers deceive users into submitting credentials to fraudulent websites designed to mimic legitimate authentication interfaces. MFA mitigates this attack by requiring a second factor that cannot be captured through the fraudulent interface.
- **Replay Attacks:** Attackers intercept valid authentication data and retransmit it to gain unauthorized access. The implemented system prevents replay attacks through single-use OTP enforcement—each OTP is invalidated immediately upon successful verification via database `used` flag.
- **Brute-Force Attacks:** Systematic enumeration of possible credentials. The system implements attempt-based rate limiting via database counter, restricting OTP verification to a maximum of three attempts before credential invalidation.
- **Session Hijacking:** Theft of authenticated session identifiers allowing attackers to impersonate legitimate users. IP address binding stored in database ensures that OTP verification requests originate from the same network location as the initial login request.

3 System Architecture

3.1 Architectural Overview

The system adheres to a client-server architectural pattern with four primary components operating in coordination:

1. **Client Interface:** Web-based user interface rendered in standard web browsers, responsible for credential collection, user interaction management, and display logic.
2. **Backend Server:** Flask-based Python application handling authentication logic, OTP lifecycle management, security validation, database operations, and communication with external services.
3. **Database Layer:** SQLite relational database (`mfa.db`) providing ACID-compliant persistent storage for user credentials and OTP records.
4. **External Communication Service:** Telegram messaging platform utilized for out-of-band OTP delivery.

The architectural design prioritizes separation of concerns, with the frontend exclusively managing presentation logic while the backend encapsulates all security-critical operations including database interactions. This separation ensures that security mechanisms cannot be bypassed through client-side manipulation, as all validation occurs on the trusted server component.

3.2 Authentication Workflow

The authentication workflow proceeds through the following sequence of interactions:

1. **Phase 1 - Initial Authentication:** The user submits username and password credentials through the web interface. The client transmits these credentials to the backend via HTTP POST to the `/login` endpoint. The backend queries the database for the user, retrieves the password hash, and validates credentials using `check_password_hash()` with constant-time comparison, implementing generic error responses to prevent username enumeration attacks.

2. **Phase 2 - OTP Generation and Delivery:** Upon successful password verification, the backend generates a cryptographically secure 6-digit OTP using Python's `secrets` module. The system invalidates any existing OTPs for the user (sets `used=1`), then inserts a new OTP record into the database with generation timestamp, expiration timestamp (created_at + 60 seconds), client IP address, and initial state (`used=0`, `attempts=0`). The backend transmits the OTP to the user's Telegram account via the Telegram Bot API.

3. **Phase 3 - OTP Verification:** The user retrieves the OTP from Telegram and submits it through a separate web interface. The backend queries the database for the latest OTP record and executes five sequential security checks: existence verification, replay prevention (`used` flag), temporal validity (compare current time vs `expires_at`), IP binding, and code validation with attempt tracking.

4. **Phase 4 - Authentication Completion:** Upon successful verification of all security checks, the OTP is marked as used (`used=1`) in the database. The record persists for audit purposes. The session is established and the user gains access to protected resources.

3.3 Technology Stack Rationale

3.3.1 Backend: Python 3.11 + Flask Framework

Python was selected for its extensive standard library support, including the `secrets` module for cryptographically secure random number generation and built-in `sqlite3` module for database operations. Flask provides a lightweight web framework with minimal boilerplate, appropriate for focused security demonstrations.

3.3.2 Database: SQLite

SQLite provides a lightweight, file-based relational database requiring no separate server process, making it ideal for academic demonstration while still offering ACID compliance, persistent storage, and relational integrity through foreign keys.

3.3.3 Password Hashing: werkzeug.security

The werkzeug.security library provides `generate_password_hash()` and `check_password_hash()` functions implementing secure password hashing with automatic salt generation and constant-time comparison to prevent timing attacks.

3.3.4 Frontend: HTML5/CSS3/JavaScript (ES6+)

Native browser technologies were selected to eliminate build tooling dependencies and maintain transparency in client-side behavior. This design decision aligns with the principle of minimizing attack surface by avoiding unnecessary complexity.

3.3.5 OOB Channel: Telegram Bot API

Telegram was selected over SMS-based alternatives due to zero per-message cost, instant delivery with higher reliability, well-documented RESTful API, and widespread availability across platforms.

3.3.6 Containerization: Docker

Docker containerization provides deployment portability across diverse hardware architectures and operating systems, enabling consistent behavior in development, testing, and deployment environments.

4 Implementation Details

4.1 Backend Architecture

The Flask backend application is structured into discrete functional modules maintaining clear separation of concerns:

- **Configuration Module:** Defines system constants including Telegram API credentials (hardcoded for academic reproducibility), database path, OTP expiration interval (60 seconds), and maximum attempt threshold (3 attempts).
- **Database Functions:** Encapsulate all SQLite operations including `get_db_connection()`, `init_db()`, and `seed_demo_user()`.
- **Route Handlers:** Four HTTP endpoints provide the complete authentication workflow:
  - `GET /`: Serves the login interface
  - `GET /otp_page`: Serves the OTP verification interface

  - `POST /login`: Processes credentials and generates OTPs
  - `POST /verify_otp`: Validates submitted OTPs through database queries

- **Helper Functions:** Encapsulate reusable logic including cryptographically secure OTP generation and Telegram API communication.
- **CORS Configuration:** The system uses open CORS (`Access-Control-Allow-Origin: *`) for demo simplicity, allowing cross-origin requests during development and testing.

4.2 Cryptographically Secure OTP Generation

The system employs Python's `secrets` module for OTP generation rather than the standard `random` module. The `secrets` module accesses the operating system's cryptographically secure random number generator (CSPRNG), providing cryptographic-quality randomness with sufficient entropy for security applications.

```python
import secrets

def generate_otp():
    """
    Generates cryptographically secure 6-digit OTP.
    Returns string in range [100000, 999999].
    """
    return str(secrets.randbelow(900000) + 100000)
```

Listing 1: Cryptographically Secure OTP Generation

4.3 SQLite Database Integration

To ensure persistent and auditable storage, the project employs SQLite as a lightweight, file-based relational database (`mfa.db`). SQLite requires no separate server process, making it suitable for academic demonstration while providing relational structure, ACID compliance, and persistence across server restarts.

4.3.1 Automatic Database Initialization

The database is created automatically on application startup through the `init_db()` function. When the server runs, it executes `CREATE TABLE IF NOT EXISTS` statements ensuring idempotent initialization:

```python
def init_db():
    """Initialize database tables on first run."""
    conn = get_db_connection()
    cursor = conn.cursor()

    # Create users table
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            username TEXT NOT NULL UNIQUE,
            password_hash TEXT NOT NULL,
            telegram_chat_id TEXT NOT NULL,
            created_at TEXT NOT NULL
        )
    ''')

    # Create otps table with foreign key to users
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS otps (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            user_id INTEGER NOT NULL,
            otp_code TEXT NOT NULL,
            created_at TEXT NOT NULL,
            expires_at TEXT NOT NULL,
            client_ip TEXT NOT NULL,
            used INTEGER NOT NULL DEFAULT 0,
            attempts INTEGER NOT NULL DEFAULT 0,
            FOREIGN KEY (user_id) REFERENCES users(id)
        )
    ''')

    conn.commit()
    conn.close()
```

Listing 2: Database Initialization

4.3.2 Database Schema

users table – Stores user identity and authentication data:

- `id`: Primary key (autoincrement)
- `username`: Unique username (UNIQUE constraint)
- `password_hash`: Hashed password via werkzeug.security
- `telegram_chat_id`: Telegram chat ID for OTP delivery
- `created_at`: Account creation timestamp (ISO 8601)

otps table – Stores OTP lifecycle and security metadata:

- `id`: Primary key (autoincrement)
- `user_id`: Foreign key to users.id
- `otp_code`: 6-digit OTP

- `created_at`: Generation timestamp
- `expires_at`: Expiration timestamp (created_at + 60s)
- `client_ip`: IP captured at login (for binding)
- `used`: Boolean flag (0=unused, 1=used) for replay prevention
- `attempts`: Failed verification counter (max 3)

4.3.3 Password Hashing

Passwords are hashed using werkzeug.security before database storage:

```python
from werkzeug.security import generate_password_hash,
    check_password_hash

# During user creation
password_hash = generate_password_hash('password123')
cursor.execute('''
    INSERT INTO users (username, password_hash, telegram_chat_id,
    created_at)
    VALUES (?, ?, ?, ?)
''', ('student', password_hash, TELEGRAM_CHAT_ID, created_at))

# During login verification (constant-time comparison)
if not check_password_hash(user['password_hash'], password):
    return jsonify({"success": False, "message": "Invalid username or
    password"}), 401
```

Listing 3: Password Hashing Implementation

4.3.4 Database-Backed Security Controls

Replay Protection: The `used` flag prevents OTP reuse. After successful verification, the system updates `used=1`:

```python
if otp_data['used'] == 1:
    return jsonify({"success": False, "message": "OTP already used"}),
    401

# After successful verification
cursor.execute('UPDATE otps SET used = 1 WHERE id = ?', (otp_data['id'
    ],))
```

Listing 4: Replay Prevention

Expiration Control: OTP verification compares current time against stored `expires_at`:

```python
expires_at = datetime.fromisoformat(otp_data['expires_at'])
if datetime.utcnow() > expires_at:
    cursor.execute('UPDATE otps SET used = 1 WHERE id = ?', (otp_data['
    id'],))
    return jsonify({"success": False, "message": "OTP expired"}), 401
```

Listing 5: Temporal Expiration

Attempt Limiting: Failed verifications increment `attempts`; at 3 failures, OTP is invalidated:

```
new_attempts = otp_data['attempts'] + 1
if new_attempts >= 3:
    cursor.execute('UPDATE otps SET used = 1, attempts = ? WHERE id = ?',
                  (new_attempts, otp_data['id'],))
    return jsonify({"success": False, "message": "Too many attempts"}), 401
```

Listing 6: Brute-Force Prevention

IP Binding: Verification checks that request IP matches stored `client_ip`:

```
client_ip = request.remote_addr
if otp_data['client_ip'] != client_ip:
    # Allow localhost variations for development
    localhost_ips = ['127.0.0.1', '::1', 'localhost']
    if not (otp_data['client_ip'] in localhost_ips and client_ip in localhost_ips):
        return jsonify({"success": False, "message": "IP mismatch"}), 403
```

Listing 7: IP Address Binding

4.4 Telegram API Integration

Communication with Telegram occurs through HTTP POST requests to the Bot API endpoint with timeout configuration (5-second timeout), comprehensive error handling, and structured message formatting.

4.5 Frontend Session Management

The frontend employs browser `sessionStorage` for temporary state management. The username is stored in `sessionStorage` after successful password verification and retrieved during OTP submission.

5 User Interface and System Demonstration

5.1 Login Interface

The system begins with a clean, professional login interface where users enter their credentials. The interface features a centered login form with username and password fields, maintaining simplicity while ensuring security.

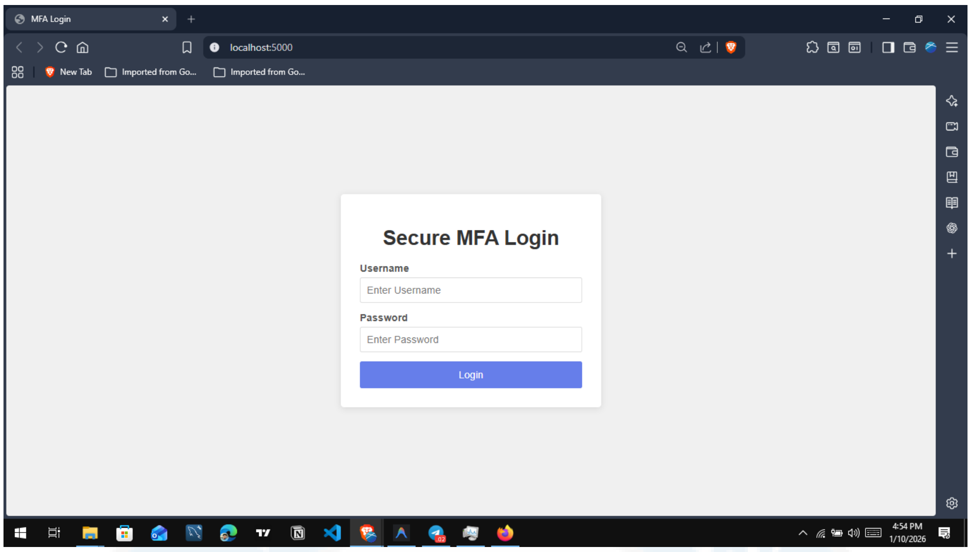


Figure 1: Secure MFA Login Interface - Users enter their credentials to begin the authentication process

The login interface implements several security features including generic error messages to prevent username enumeration, client-side input validation, and secure transmission of credentials to the backend server.

5.2 OTP Verification Interface

After successful password verification, users are redirected to the OTP verification page. This interface clearly communicates the next steps and displays the 60-second validity window for the OTP.

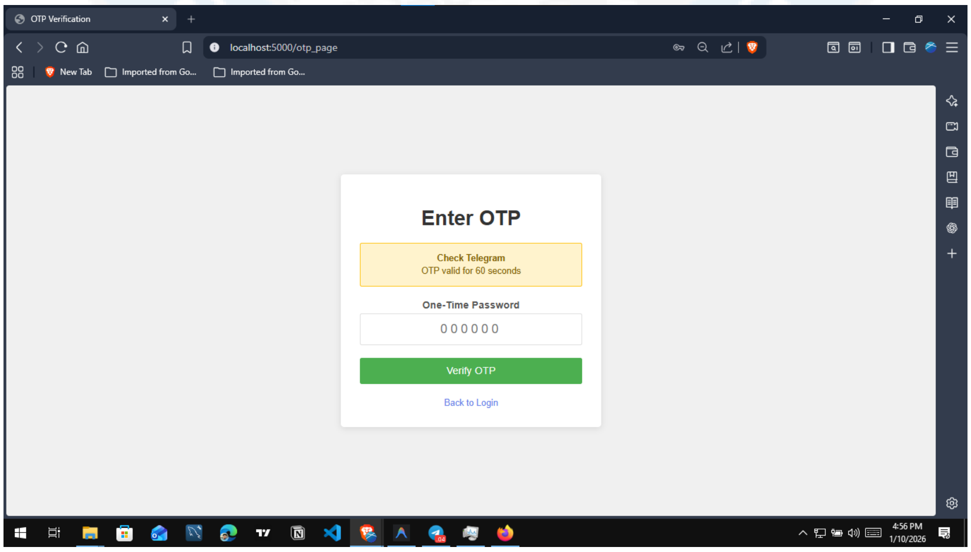
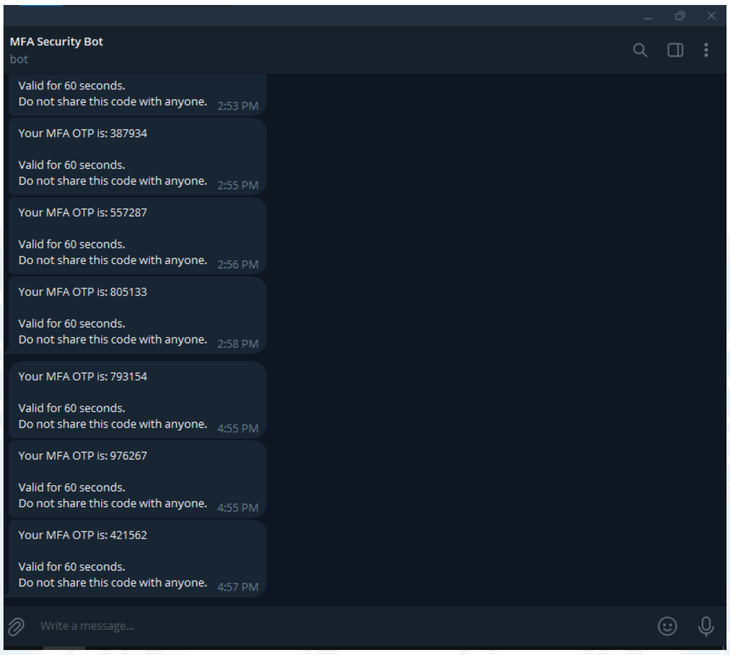


Figure 2: OTP Verification Page - Users enter the 6-digit code received via Telegram

The interface includes a prominent notification reminding users to check their Telegram messages, displays the time-limited nature of the OTP, and provides a convenient "Back to Login" option for users who need to restart the authentication process.

5.3 Telegram OTP Delivery

OTPs are delivered instantly to users via the Telegram messaging platform. Each message includes the 6-digit code, validity information, and security warnings.



Valid for 60 seconds.
Do not share this code with anyone. 2:53 PM

Your MFA OTP is: 387934

Valid for 60 seconds.
Do not share this code with anyone. 2:55 PM

Your MFA OTP is: 557287

Valid for 60 seconds.
Do not share this code with anyone. 2:56 PM

Your MFA OTP is: 805133

Valid for 60 seconds.
Do not share this code with anyone. 2:58 PM

Your MFA OTP is: 793154

Valid for 60 seconds.
Do not share this code with anyone. 4:55 PM

Your MFA OTP is: 976267

Valid for 60 seconds.
Do not share this code with anyone. 4:55 PM

Your MFA OTP is: 421562

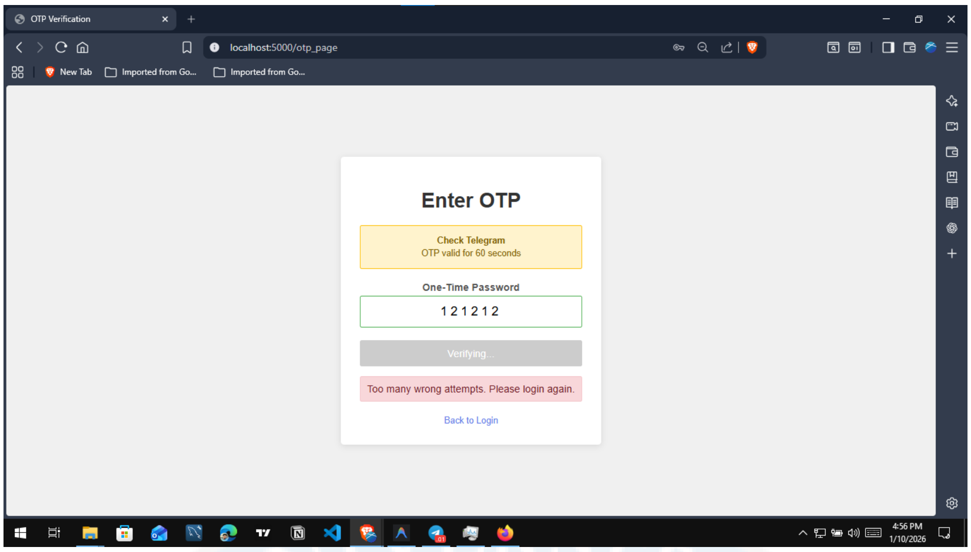
Valid for 60 seconds.
Do not share this code with anyone. 4:57 PM

Figure 3: Telegram Bot delivering OTPs with security warnings and validity information

The Telegram integration provides several benefits including instant delivery notification, persistent message history for reference, and platform-independent access across mobile and desktop devices.

5.4 Failed Attempt Handling

The system provides clear feedback when incorrect OTPs are entered, displaying remaining attempts and appropriate error messages.


Enter OTP

Check Telegram
OTP valid for 60 seconds

One-Time Password
121212

Verifying...

Too many wrong attempts. Please login again.

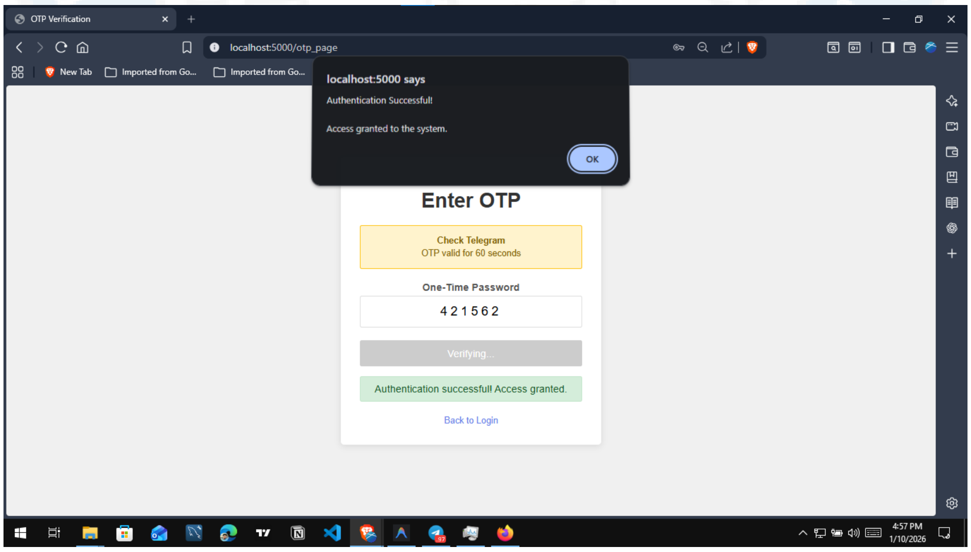
Back to Login

Figure 4: Failed OTP verification showing attempt counter and error messaging

After three failed attempts, users are automatically redirected to the login page and must complete the entire authentication process again, including password verification and new OTP generation.

5.5 Successful Authentication

Upon successful OTP verification, users receive immediate confirmation and are granted system access.


localhost:5000 says
Authentication Successful!

Access granted to the system.

OK

Enter OTP

Check Telegram
OTP valid for 60 seconds

One-Time Password
421562

Verifying...

Authentication successful! Access granted.

Back to Login

Figure 5: Successful authentication confirmation with access granted message

6 Security Mechanisms and Attack Mitigation

6.1 The Five Sequential Security Checks

OTP verification executes five discrete validation checks in sequence. Each check addresses specific attack vectors and may result in immediate authentication failure.

6.1.1 Check 1: Existence Verification

Purpose: Ensures OTP verification corresponds to a valid, initiated login session.

Attack Mitigated: Prevents unauthorized access attempts where attackers bypass the login phase.

6.1.2 Check 2: Replay Attack Prevention

Purpose: Ensures each OTP can only be successfully verified once.

Attack Mitigated: Prevents replay attacks where attackers intercept legitimate OTP transmissions and attempt reuse.

6.1.3 Check 3: Temporal Validity Enforcement

Purpose: Enforces a 60-second validity window for OTPs.

Attack Mitigated: Limits the window of opportunity for OTP interception and use.

6.1.4 Check 4: IP Address Binding

Purpose: Ensures OTP verification originates from the same IP address as the initial login request.

Attack Mitigated: Prevents session hijacking scenarios where attackers steal credentials but attempt use from different network locations.

6.1.5 Check 5: Cryptographic Code Validation

Purpose: Validates the submitted OTP using constant-time comparison.

Attack Mitigated: The use of `secrets.compare_digest()` prevents timing-based side-channel attacks.

6.2 Attempt-Based Brute-Force Mitigation

The system implements a strict three-attempt limit for OTP verification. With 1,000,000 possible 6-digit combinations and only three attempts permitted, the probability of successful brute-force attack is 0.0003%.

6.3 Out-of-Band Delivery Security Advantages

Delivering OTPs via Telegram provides several security advantages:

- **Channel Independence:** The web application and Telegram operate on entirely separate infrastructure
- **Physical Device Requirement:** Attackers must possess the specific physical device
- **Intrusion Detection:** Unsolicited OTP messages serve as implicit alerts

7 Threat Model and Security Analysis

7.1 Attacker Assumptions

The threat model assumes the following attacker capabilities:

Attacker Capabilities:

- Network access to the web application
- Ability to intercept and analyze network traffic
- Potential knowledge of valid usernames
- Access to automated credential enumeration tools

Attacker Constraints:

- Does not possess the user's physical Telegram device
- Cannot compromise Telegram infrastructure directly
- Limited to network-based attacks

7.2 Attack Vector Analysis

7.2.1 Brute-Force Attack on OTP

System Response: Three-attempt limit renders brute-force attacks statistically infeasible.

Effectiveness: HIGH - Attack effectively mitigated.

7.2.2 OTP Interception

System Response: 60-second expiration window limits utility of intercepted OTPs.

Effectiveness: MEDIUM - Limited time window reduces but does not eliminate risk.

7.2.3 Replay Attack

System Response: Single-use enforcement prevents credential reuse.
Effectiveness: HIGH - Replay attacks completely prevented.

7.2.4 Session Hijacking

System Response: IP address binding prevents remote hijacking.
Effectiveness: MEDIUM - Effective against remote attackers.

7.2.5 Credential Phishing

System Response: Two-factor requirement protects against password capture.
Effectiveness: HIGH - Substantially elevates attack complexity.

8 Testing and Validation

8.1 Testing Methodology

Comprehensive manual testing was conducted to validate all functional and security requirements. Testing proceeded systematically through positive cases, negative cases, and edge cases.

8.2 Functional Test Cases

Table 1: Functional Test Cases and Results

Test ID	**Procedure**	**Expected Result**	**Status**
F-01	Submit valid credentials, retrieve OTP, submit within 60 seconds	Authentication succeeds	PASS
F-02	Submit incorrect password	Generic error, no OTP	PASS
F-03	Submit non-existent username	Same generic error	PASS
F-04	Direct navigation to /otp_page without session	OTP verification fails without valid session data	PASS

8.3 Security Test Cases

Table 2: Security Test Cases and Results

Test ID	Procedure	Expected Result	Status
S-01	Wait 61 seconds before OTP entry	Expiration error	PASS
S-02	Attempt OTP reuse	Already used error	PASS
S-03	Three incorrect attempts	OTP invalidation	PASS
S-04	Verification from different IP	IP mismatch error	LIMITED
S-05	OTP verification without valid session	Server-side validation prevents unauthorized access	PASS

9 Deployment and Portability

9.1 Local Execution

The system supports direct execution on local development machines with minimal dependencies:

```
cd backend
pip install -r requirements.txt
python app.py
# Database (mfa.db) is created automatically on first run
# Demo user student/password123 is seeded with hashed password
```

Listing 8: Local Deployment Commands

9.2 Docker Containerization

The system includes a `Dockerfile` specifying the complete runtime environment. Containerization provides environment consistency, isolation, portability, and reproducibility.

9.3 Raspberry Pi and ARM Compatibility

The containerization approach demonstrates system suitability for ARM-based systems including Raspberry Pi. The `python:3.11-slim` base image supports multiple architectures including `linux/arm64`.

10 Limitations and Scope Boundaries

10.1 Acknowledged Design Limitations

The following limitations are explicitly acknowledged:

10.1.1 HTTP-Only Communication

Limitation: All communication over HTTP without TLS/SSL.
Impact: Credentials vulnerable to network interception.
Production Solution: Deploy behind HTTPS with valid TLS certificates.

10.1.2 Flask Development Server

Limitation: Uses Flask's built-in development server.
Impact: Limited concurrency, not production-ready.
Production Solution: Deploy with Gunicorn or uWSGI.

10.1.3 Hardcoded Telegram Credentials

Limitation: Telegram Bot token and chat ID hardcoded in source code for academic reproducibility.
Impact: Credentials visible in source code.
Production Solution: Use environment variables or secrets management systems.

10.2 Scope Justification

These limitations represent intentional design decisions reflecting the academic context. The focus on core authentication security principles enables concentrated examination of fundamental concepts without production infrastructure complexity.

11 Future Enhancements

11.1 Database Scalability

While SQLite provides adequate performance for academic demonstrations, production systems with high concurrency would benefit from migration to client-server databases:

- PostgreSQL for advanced features and horizontal scaling
- MySQL for proven scalability in high-traffic applications
- Redis for high-performance OTP storage with native TTL support

11.2 Advanced Rate Limiting

Implementation of distributed rate limiting using Redis would provide protection against coordinated attacks from distributed sources.

11.3 Multiple Authentication Channels

Expansion beyond Telegram to support alternative OTP delivery:

- SMS delivery via Twilio or Amazon SNS
- Email-based OTP delivery
- TOTP support for authenticator apps
- FIDO2/WebAuthn hardware security keys

11.4 Enhanced Session Management

Implementation of secure session tokens using cryptographically signed cookies with appropriate security flags.

12 Project Achievements and Learning Outcomes

12.1 Technical Skills Development

Working on this Multi-Factor Authentication System helped me develop many important technical skills in security engineering and web application development. I learned how to implement cryptographic primitives correctly using Python's `secrets` module, which taught me the critical difference between statistical randomness and cryptographic security.

The project provided hands-on experience with secure authentication workflows, teaching me how to design systems that resist common attack vectors. I gained practical understanding of session management, temporal constraints in security contexts, database integration for persistent storage, password hashing best practices, and the importance of defense-in-depth architectures where multiple independent mechanisms work together to provide comprehensive protection.

I also developed proficiency with the Flask web framework, learning how to structure backend applications for security-critical contexts. The integration with Telegram's Bot API taught me about working with external services, handling API errors gracefully, and implementing timeout mechanisms to prevent indefinite blocking. Database design and SQL operations deepened my understanding of relational data modeling and ACID compliance.

12.2 Security Analysis and Threat Modeling

This project taught me how to think like both a system designer and a potential attacker. I learned to systematically analyze threat models, identifying what capabilities attackers might have and what constraints they operate under. This perspective helped me design security mechanisms that address realistic threats rather than theoretical vulnerabilities that rarely occur in practice.

The process of implementing five sequential security checks taught me about layered security and how each mechanism addresses specific attack vectors. I learned that effective security comes not from a single strong mechanism but from multiple complementary defenses that collectively make successful attacks impractical.

Testing the security mechanisms through deliberate attack simulations was particularly educational. By attempting to break my own system through brute-force attacks, replay attempts, and session hijacking, I gained confidence that the security measures work as intended while identifying edge cases that needed additional handling.

12.3 System Design and Architecture

The project reinforced fundamental software engineering principles including separation of concerns, modular design, and clear interfaces between components. I learned how to structure applications so that security-critical logic remains isolated on the server side where it cannot be tampered with by clients.

Docker containerization taught me about deployment portability and environment consistency. Understanding how to create reproducible environments that work identically across different hardware platforms, including ARM-based systems like Raspberry Pi, demonstrated the practical value of containerization for modern application deployment.

12.4 Problem-Solving and Debugging

Throughout development, I encountered numerous challenges that required systematic problem-solving. When the Telegram API integration initially failed, I learned to read API documentation carefully, implement proper error handling, and use timeout mechanisms to prevent application hangs. When timing issues arose with OTP expiration, I learned about precision in temporal calculations and the importance of consistent time measurement.

Debugging security mechanisms required particular care because errors could either create vulnerabilities or unnecessarily restrict legitimate users. I learned to balance security with usability, ensuring the system remains secure without creating frustrating user experiences.

12.5 Professional Development

This project taught me to maintain clear documentation throughout the development process. Writing this comprehensive report helped me practice technical writing, learning to explain complex security concepts clearly while maintaining technical accuracy.

I developed appreciation for the complexity involved in building secure systems and the importance of testing every security mechanism thoroughly. The experience showed me that security is not something that can be added as an afterthought but must be designed into the system from the beginning.

13 Conclusion

This project successfully designed, implemented, and analyzed a complete Multi-Factor Authentication system demonstrating fundamental security principles in practical application. The implemented system combines password-based authentication (with werkzeug.security hashed passwords stored in SQLite database) with time-limited One-Time Passwords delivered through independent communication channels, substantially elevating the difficulty threshold for unauthorized access beyond single-factor alternatives.

The authentication workflow incorporates five discrete security validation checks addressing specific attack vectors: existence verification prevents unauthorized session access, replay prevention eliminates credential reuse via database `used` flag, temporal constraints limit attack windows through stored expiration timestamps, IP binding impedes remote session hijacking, and cryptographic code validation with database-tracked attempts resists timing-based analysis and brute-force enumeration. The three-attempt limit on OTP verification provides robust brute-force protection, reducing the probability of successful attack to 0.0003% within the 60-second validity window.

Implementation employed modern security practices including cryptographically secure random number generation through Python's `secrets` module, constant-time comparison algorithms preventing side-channel attacks, password hashing via werkzeug.security, SQLite database providing ACID-compliant persistent storage with automatic schema initialization, and out-of-band communication through the Telegram platform. The system demonstrates platform independence through Docker containerization, enabling deployment across diverse hardware architectures including ARM-based embedded systems.

Testing validated all functional requirements and security mechanisms, confirming successful mitigation of common attack vectors including credential replay, brute-force enumeration, and session hijacking attempts. The systematic testing methodology encompassed positive cases, negative cases, and edge conditions, providing comprehensive validation of system behavior.

This work contributes to practical understanding of authentication security, demonstrating how theoretical security concepts manifest in functional implementations. The project reinforces fundamental principles including defense-in-depth through multiple independent validation layers, principle of least privilege through minimal credential validity windows, security through isolation via out-of-band communication channels, and data persistence through relational database integration.

The experience of developing this system from initial concept through implementation, database integration, testing, and documentation has significantly enhanced my capabilities as a software engineer with security focus. The skills and knowledge gained through this project provide a strong foundation for future work in secure system design and implementation.

Acknowledgments

I am deeply grateful to the Università degli Studi di Messina for providing excellent educational opportunities and resources that made this project possible. The comprehensive coursework in System Security provided the theoretical foundation necessary to undertake this practical implementation.

I would like to extend special thanks to Prof. Massimo Villari and Prof. Armando Ruggeri for their outstanding teaching, guidance, and mentorship throughout my studies. Their expertise in system security and distributed computing has been instrumental in developing the skills and knowledge that enabled me to complete this project successfully. Their encouragement to explore practical implementations of security concepts helped me bridge the gap between theoretical understanding and real-world application.

I am also thankful to my fellow students who provided valuable discussions about security concepts and implementation challenges. These conversations helped refine my understanding and identify edge cases I might have otherwise overlooked.

Finally, I acknowledge the open-source community whose excellent documentation and tools made this implementation possible, including the Python Software Foundation, the Flask development team, and Telegram for providing their Bot API platform.

References

[1] Verizon. (2023). *Data Breach Investigations Report.* Verizon Enterprise Solutions.

[2] OWASP Foundation. (2021). *Authentication Cheat Sheet.* OWASP Cheat Sheet Series. Retrieved from https://cheatsheetseries.owasp.org/

[3] Anderson, R. (2020). *Security Engineering: A Guide to Building Dependable Distributed Systems* (3rd ed.). Wiley.

[4] Grassi, P. A., Garcia, M. E., & Fenton, J. L. (2017). *Digital Identity Guidelines: Authentication and Lifecycle Management.* NIST Special Publication 800-63B. National Institute of Standards and Technology.

[5] M'Raihi, D., Machani, S., Pei, M., & Rydell, J. (2011). *TOTP: Time-Based One-Time Password Algorithm.* RFC 6238. Internet Engineering Task Force (IETF).

[6] M'Raihi, D., Bellare, M., Hoornaert, F., Naccache, D., & Ranen, O. (2005). *HOTP: An HMAC-Based One-Time Password Algorithm.* RFC 4226. Internet Engineering Task Force (IETF).

[7] Stallings, W., & Brown, L. (2018). *Computer Security: Principles and Practice* (4th ed.). Pearson Education.

[8] Schneier, B. (2015). *Applied Cryptography: Protocols, Algorithms, and Source Code in C* (20th Anniversary Edition). Wiley.

[9] Bonneau, J., Herley, C., Van Oorschot, P. C., & Stajano, F. (2012). The Quest to Replace Passwords: A Framework for Comparative Evaluation of Web Authentication Schemes. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy* (pp. 553-567). IEEE.

A User Guide and System Setup

A.1 System Requirements

To run this Multi-Factor Authentication System, you need:

- Computer with Windows, macOS, or Linux
- Python 3.11 or higher installed
- Internet connection for Telegram API communication
- Telegram account for receiving OTPs
- Web browser (Chrome, Firefox, Edge, or Safari)

For Docker deployment:

- Docker Engine installed
- 512MB RAM minimum
- Compatible with ARM64 architecture (Raspberry Pi 4/5)

A.2 Installation and Setup

A.2.1 Local Installation

1. Clone or download the project repository
2. Navigate to the backend directory
3. Install required Python packages:
```
pip install -r requirements.txt
```
4. Run the application:
```
python app.py
```
5. The SQLite database (`mfa.db`) will be automatically created in the project root
6. A demo user (`student/password123`) will be automatically seeded with hashed password
7. Open browser to `http://localhost:5000`

A.3 Docker Deployment

1. Build the Docker image:

```
docker build -t mfa-system .
```

2. Run the container:

```
docker run -p 5000:5000 mfa-system
```

3. Access at `http://localhost:5000`

A.4 Using the System

A.4.1 Initial Login

1. Navigate to the login page
2. Enter username: `student`
3. Enter password: `password123`
4. Click "Login" button
5. Wait for OTP to be sent to your Telegram account

A.4.2 OTP Verification

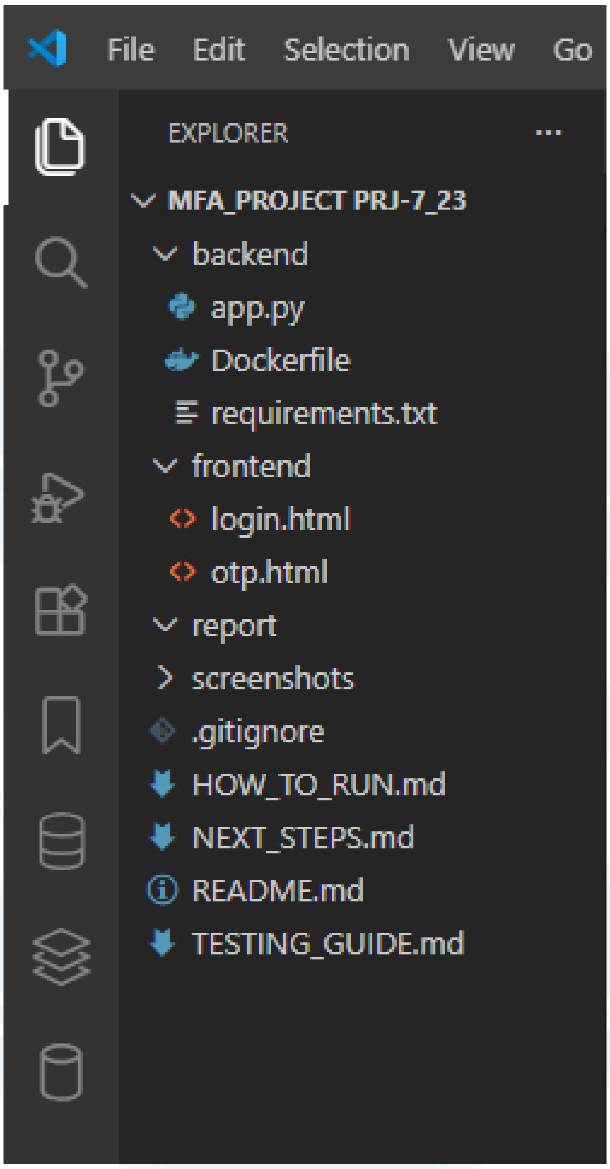
1. Check your Telegram messages for the OTP
2. Note that the code is valid for only 60 seconds
3. Enter the 6-digit code in the verification page
4. Click "Verify OTP"
5. You have 3 attempts to enter the correct code

A.4.3 Security Recommendations

- Never share your OTP codes with anyone
- Complete OTP verification within 60 seconds
- If you receive unsolicited OTP messages, someone may be attempting to access your account
- Log out when finished using the system
- Use a strong, unique password

B Code Repository Structure

The project follows a well-organized directory structure for clear separation of concerns and easy maintenance:



```
MFA_PROJECT PRJ-7_23
  backend
    app.py
    Dockerfile
    requirements.txt
  frontend
    login.html
    otp.html
  report
  screenshots
  .gitignore
  HOW_TO_RUN.md
  NEXT_STEPS.md
  README.md
  TESTING_GUIDE.md
```

Figure 6: Project folder structure showing organized backend, frontend, and configuration files

The main components include:

- **backend/** - Contains the Flask application (`app.py`), Docker configuration, and Python dependencies
- **frontend/** - Houses the HTML interfaces for login and OTP verification
- **mfa.db** - SQLite database file (auto-created on first run)
- **report/** - Documentation including screenshots and LaTeX source files
- **Configuration files** - README, testing guides, and setup instructions

C Configuration Options

Key configuration parameters in `app.py`:

```python
# Telegram Bot Configuration (hardcoded for demo reproducibility)
TELEGRAM_BOT_TOKEN = "8599518592:AAHHY0WU2ZtK3i2W9bemkKDVZc1x3fUAIv8"
TELEGRAM_CHAT_ID = "1415538518"

# Database Path
DATABASE_PATH = os.path.join(PROJECT_ROOT, 'mfa.db')

# OTP Configuration
OTP_EXPIRATION_TIME = 60  # seconds
MAX_OTP_ATTEMPTS = 3        # maximum failed attempts

# CORS (open for demo simplicity)
@app.after_request
def after_request(response):
    response.headers.add('Access-Control-Allow-Origin', '*')
    return response
```

D Troubleshooting Guide

D.1 Common Issues and Solutions

Issue: OTP not received in Telegram.

Solutions: Verify Telegram Bot token is correct, check Chat ID configuration, ensure internet connectivity, verify Telegram Bot is not blocked.

Issue: ”OTP expired” message.

Solutions: Complete verification within 60 seconds, return to login page and start again, check system clock synchronization.

Issue: ”IP address mismatch” error.

Solutions: Complete entire process from same device, avoid VPN changes during authentication, use consistent network connection.

Issue: Database not created.

Solutions: Check write permissions in project directory, verify Python has sqlite3 module, check console for `init_db()` success message.