Secure Authentication System - Implementation Guide

🔒 Security Features Implemented

1. **Authentication & Authorization**

- JWT-based authentication with access and refresh tokens
- Role-based access control (RBAC)
- Two-factor authentication (2FA)
- Temporary tokens for 2FA flow
- Token expiration and refresh mechanism

2. **Password Security**

- Password hashing (SHA256 for demo, use bcrypt/argon2 in production)
- Strong password requirements
- No password transmission in plain text

3. **Account Security**

- Account lockout after failed attempts (5 attempts = 15 min lockout)
- Failed attempt tracking
- 2FA code replay attack prevention
- Session management

4. **API Security**

- CORS configuration
- Security headers (X-Frame-Options, CSP, etc.)
- Bearer token authentication
- Request validation with Pydantic
- Input sanitization to prevent injection

5. **Token Security**

- Short-lived access tokens (15 minutes)
- Longer-lived refresh tokens (7 days)
- Token type validation

- JWT ID (jti) for token tracking
- Secure token storage recommendations

📦 Installation & Setup

Requirements

```bash
pip install fastapi uvicorn python-jose[cryptography] python-multipart pydantic
```

Required packages (requirements.txt):

```
fastapi==0.104.1
uvicorn==0.24.0
python-jose[cryptography]==3.3.0
python-multipart==0.0.6
pydantic==2.4.2
passlib[bcrypt]==1.7.4  # For production password hashing
pyotp==2.9.0  # For production 2FA
redis==5.0.1  # For production session storage
```

Running the API

```bash
# Development
uvicorn secure_api:app --reload --port 8000

# Production with SSL
uvicorn secure_api:app --host 0.0.0.0 --port 443 --ssl-keyfile=./key.pem --ssl-certfile=./cert.pem
```

🚨 Security Best Practices

1. Environment Variables

Never hardcode secrets! Use environment variables:

```python
```

```python
import os
from dotenv import load_dotenv

load_dotenv()

SECRET_KEY = os.getenv("SECRET_KEY")
DATABASE_URL = os.getenv("DATABASE_URL")
```

2. Password Hashing

Replace SHA256 with bcrypt or argon2:

```python
from passlib.context import CryptContext

pwd_context = CryptContext(schemes=["bcrypt"], deprecated="auto")

def hash_password(password: str) -> str:
    return pwd_context.hash(password)

def verify_password(plain_password: str, hashed_password: str) -> bool:
    return pwd_context.verify(plain_password, hashed_password)
```

3. Real 2FA Implementation

Use TOTP with pyotp:

```python
import pyotp

def generate_totp_secret():
    return pyotp.random_base32()

def verify_totp(secret: str, code: str) -> bool:
    totp = pyotp.TOTP(secret)
    return totp.verify(code, valid_window=1)
```

4. Database Storage

Replace in-memory storage with a database:

```python
from sqlalchemy import create_engine
from sqlalchemy.orm import sessionmaker

# Use PostgreSQL or MySQL in production
engine = create_engine("postgresql://user:pass@localhost/db")
SessionLocal = sessionmaker(bind=engine)
```

5. **Redis for Sessions**

Use Redis for token blacklisting and session management:

```python
import redis

redis_client = redis.Redis(host='localhost', port=6379, decode_responses=True)

def blacklist_token(token: str, expiry: int):
    redis_client.setex(f"blacklist:{token}", expiry, "1")

def is_token_blacklisted(token: str) -> bool:
    return redis_client.exists(f"blacklist:{token}")
```

🛡️ Security Vulnerabilities to Address

Critical Issues for Production:

1. **HTTPS Only**
   - Always use SSL/TLS in production
   - Enforce HTTPS with HSTS headers
   - Use secure cookies with `Secure` and `HttpOnly` flags
2. **Rate Limiting**

   ```python
   ```

```python
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)

@app.post("/api/auth/login")
@limiter.limit("5/minute")
async def login(request: LoginRequest):
    # Login logic
```

3. **CSRF Protection**
   - Implement CSRF tokens for state-changing operations
   - Use SameSite cookie attribute
   - Validate Referer/Origin headers
4. **SQL Injection Prevention**
   - Use parameterized queries
   - Validate all inputs
   - Use an ORM like SQLAlchemy
5. **XSS Prevention**
   - Sanitize all user inputs
   - Use Content Security Policy headers
   - Escape output in templates

🔐 Frontend Security Considerations

Token Storage Options:

1. **Session Storage** (Current - Good)
   - Cleared when tab closes
   - Not accessible across tabs
   - Protected from XSS with proper CSP
2. **HttpOnly Cookies** (Best)
   - Not accessible via JavaScript
   - Protected from XSS
   - Requires CSRF protection
3. **Never Use localStorage**

- Persists across sessions
- Vulnerable to XSS attacks

Secure API Calls:

```javascript
// Always include security headers
const secureHeaders = {
  'X-Requested-With': 'XMLHttpRequest',
  'Content-Type': 'application/json'
};

// Use credentials for cookies
fetch(url, {
  credentials: 'include',
  headers: secureHeaders
});
```

📊 Security Monitoring

Implement Logging:

```python
import logging
from datetime import datetime

logger = logging.getLogger(__name__)

@app.post("/api/auth/login")
async def login(request: LoginRequest):
  logger.info(f"Login attempt for user: {request.username} at {datetime.utcnow()}")
  # Log security events
  if failed_attempt:
    logger.warning(f"Failed login for {request.username} from {request_ip}")
```

Security Metrics to Track:

- Failed login attempts
- Account lockouts
- Token refresh patterns

- Unusual access patterns
- 2FA bypass attempts
- API rate limit hits

🚀 Production Checklist

- [ ] Use environment variables for all secrets
- [ ] Implement proper password hashing (bcrypt/argon2)
- [ ] Set up real TOTP-based 2FA
- [ ] Use PostgreSQL/MySQL instead of in-memory storage
- [ ] Implement Redis for session management
- [ ] Add rate limiting to all endpoints
- [ ] Set up HTTPS with valid SSL certificates
- [ ] Configure security headers properly
- [ ] Implement comprehensive logging
- [ ] Set up monitoring and alerting
- [ ] Regular security audits
- [ ] Implement CSRF protection
- [ ] Use HttpOnly cookies for tokens
- [ ] Validate and sanitize all inputs
- [ ] Implement proper error handling
- [ ] Set up backup and recovery procedures
- [ ] Document API security measures
- [ ] Train team on security best practices

🔍 Testing Security

Tools for Security Testing:

1. **OWASP ZAP** - Web application security scanner
2. **Burp Suite** - Security testing toolkit
3. **SQLMap** - SQL injection testing
4. **Postman** - API testing with security checks
5. **pytest** - Unit testing for security functions

Example Security Tests:

```python
```

```python
import pytest
from fastapi.testclient import TestClient

def test_sql_injection_protection():
    response = client.post("/api/auth/login", json={
        "username": "admin' OR '1'='1",
        "password": "password"
    })
    assert response.status_code == 422  # Validation error

def test_rate_limiting():
    for i in range(6):
        response = client.post("/api/auth/login", json={
            "username": "test",
            "password": "wrong"
        })
    assert response.status_code == 429  # Too many requests

def test_token_expiration():
    # Create expired token
    expired_token = create_token({"sub": "test"}, timedelta(seconds=-1))
    response = client.get("/api/protected/data",
                          headers={"Authorization": f"Bearer {expired_token}"})
    assert response.status_code == 401
```

📚 Additional Resources

- OWASP Top 10
- FastAPI Security Documentation
- JWT Best Practices
- NIST Authentication Guidelines
- Mozilla Web Security Guidelines