PassOP

Secure Password Manager

Project Number 6: Password Hashing with Salt

Name: Ajitesh Kumar Singh

Roll Number: IMT2022559

Project Report

October 27, 2025

GitHub Repository:
https://github.com/Transyltooniaa/PasswordManager.git

Contents

1 Introduction

1.1 Project Overview

PassOP is a comprehensive password management solution that enables users to securely store and manage website credentials. The application combines a modern React frontend with a robust Node.js backend, all backed by MongoDB for data persistence. The core focus of the project is security, achieved through authenticated encryption-at-rest using AES-256-GCM with native Node.js crypto libraries.

1.2 Motivation

In today's digital landscape, managing multiple passwords across various platforms has become increasingly challenging. PassOP addresses this need by providing a secure, self-hosted solution that gives users complete control over their sensitive credential data while maintaining ease of use and accessibility.

1.3 Key Features

- **Secure Storage**: All passwords encrypted at rest using AES-256-GCM
- **Master Password Protection**: Single master password with bcrypt hashing
- **Search and Filter**: Quick credential lookup by site, username, or tags
- **Auto-lock**: Automatic session termination on inactivity
- **Favicon Integration**: Visual site identification
- **Tag Management**: Organize credentials with custom tags
- **Responsive Design**: Works seamlessly across devices
- **Docker Support**: Easy deployment with containerization

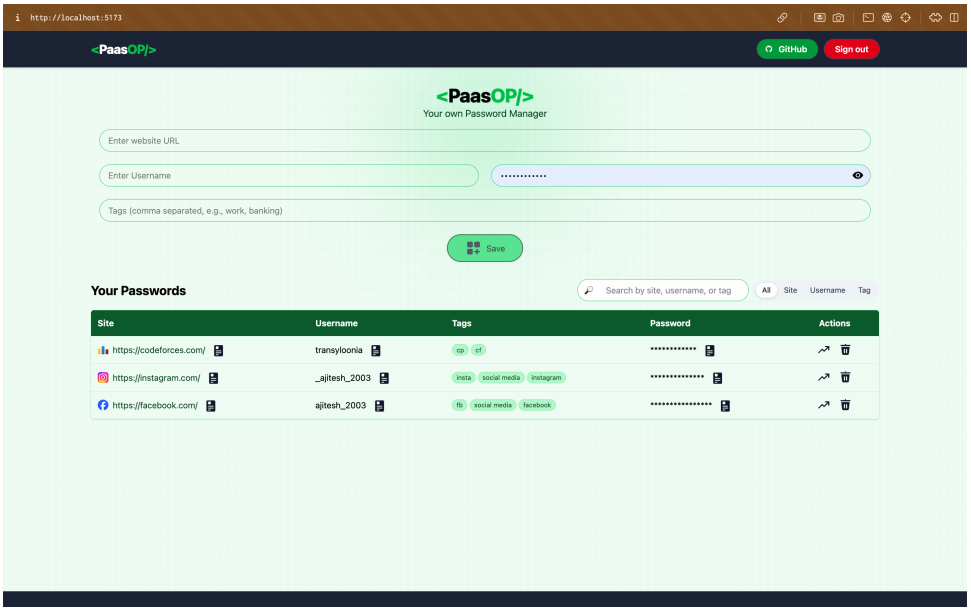

Figure 1: PassOP Main Interface

2 System Architecture

2.1 Technology Stack

2.1.1 Frontend

- **React 18**: Modern UI framework with hooks
- **Vite**: Fast build tool and development server
- **Tailwind CSS**: Utility-first CSS framework
- **Framer Motion**: Animation library for smooth transitions

2.1.2 Backend

- **Node.js 20**: JavaScript runtime environment
- **Express.js**: Web application framework
- **MongoDB**: NoSQL database for document storage
- **JWT**: JSON Web Tokens for stateless authentication
- **bcrypt**: Password hashing algorithm
- **Native Crypto**: Node.js crypto module for encryption

2.1.3 DevOps

- **Docker**: Containerization platform
- **Docker Compose**: Multi-container orchestration
- **Nginx**: Web server and reverse proxy

2.2 Component Interaction

The frontend communicates with the backend through RESTful APIs over HTTPS. All sensitive operations require JWT authentication tokens obtained after master password validation. The backend handles all cryptographic operations, ensuring passwords are encrypted before storage and decrypted only when requested by authenticated clients.

3 Security Design

3.1 Threat Model

PassOP is designed to protect against the following threats:

- Unauthorized access to stored credentials
- Database compromise or data breach
- Session hijacking and replay attacks
- Man-in-the-middle attacks
- Brute force password attempts

3.2 Authentication Mechanism

3.2.1 Master Password Protection

The application uses a single master password to control access. This password is never stored in plaintext; instead, it is hashed using bcrypt with a configurable cost factor.

```
AUTH_PASSWORD_HASH=<bcrypt_hash>
AUTH_SECRET=<jwt_signing_secret>
```

Listing 1: Environment Configuration for Authentication

3.2.2 JWT Session Management

Upon successful authentication, the server issues a JSON Web Token (JWT) with the following characteristics:

- 12-hour expiration time
- Signed with HMAC SHA-256
- Stored in browser localStorage
- Required for all API operations

3.3 Encryption Implementation

3.3.1 Algorithm Selection

PassOP employs AES-256-GCM (Galois/Counter Mode) for authenticated encryption, providing both confidentiality and integrity protection.

Key Parameters:

- **Key Size**: 256 bits (32 bytes)
- **IV Size**: 96 bits (12 bytes)
- **Auth Tag**: 128 bits (16 bytes)
- **Block Cipher**: AES

3.3.2 Key Derivation

The encryption key is obtained through the following priority order:

1. **ENCRYPTION_KEY**: Direct base64-encoded 256-bit key (recommended)
2. **Derived Key**: PBKDF2 derivation from SECRET or AUTH_SECRET
    - Iterations: 100,000
    - Hash function: SHA-512
    - Salt: Deterministic (for key stability)
    - Output: 32 bytes

3.3.3 Encryption Process

1. Generate random 12-byte initialization vector (IV)
2. Create AES-256-GCM cipher with key and IV
3. Encrypt plaintext password
4. Extract 16-byte authentication tag
5. Store IV, tag, and ciphertext in database

```
function encrypt(plaintext, { key }) {
  const iv = crypto.randomBytes(12);
  const cipher = crypto.createCipheriv('aes-256-gcm', key, iv);
  const encrypted = Buffer.concat([
    cipher.update(plaintext, 'utf8'),
    cipher.final()
  ]);
  const tag = cipher.getAuthTag();
  return {
    enc: 'aes-256-gcm',
    iv: iv.toString('base64'),
    tag: tag.toString('base64'),
    ct: encrypted.toString('base64')
  };
}
```

Listing 2: Encryption Function Structure

3.4 Security Best Practices

- No external cryptographic libraries (auditability)
- Random IV generation for each encryption operation
- Authentication tag verification on decryption
- Constant-time comparisons for tag validation
- Never store plaintext passwords in database
- Environment-based secret management
- Auto-lock on user inactivity

4 Data Model and Database Design

4.1 Database Selection

MongoDB was chosen for its flexibility in handling JSON-like documents, ease of deployment, and native Node.js driver support.

4.2 Primary Collection: passwords

The application uses a single collection to store encrypted credential records. Each document follows this schema:

```
{
  id: "uuid-v4-string",
  site: "https://example.com",
  username: "user@example.com",
  password: {
    enc: "aes-256-gcm",
    iv: "base64-encoded-12-bytes",
    tag: "base64-encoded-16-bytes",
    ct: "base64-encoded-ciphertext"
  },
  tags: ["work", "email"],
  icon: "https://example.com/favicon.ico",
  createdAt: ISODate("2025-10-27T..."),
  updatedAt: ISODate("2025-10-27T...")
}
```

Listing 3: Password Document Schema

4.3 Field Descriptions

Field	Type	Description
id	String	Client-generated UUID for unique identification
site	String	Normalized URL with protocol (https://)
username	String	Account username or email
password	Object	Encrypted password with metadata
password.enc	String	Encryption algorithm identifier
password.iv	String	Base64-encoded initialization vector
password.tag	String	Base64-encoded authentication tag
password.ct	String	Base64-encoded ciphertext
tags	Array	Categorization tags for organization
icon	String	Favicon URL for visual identification
createdAt	Date	Document creation timestamp
updatedAt	Date	Last modification timestamp

Table 1: Password Collection Schema

4.4 Data Flow

4.4.1 Write Operations (Create/Update)

1. User submits credential data through frontend
2. Backend controller normalizes site URL
3. Password field is encrypted using AES-256-GCM
4. Encrypted object stored in database
5. Success response returned to client

4.4.2 Read Operations (List/Retrieve)

1. Client requests credentials with JWT token
2. Backend queries MongoDB collection
3. Each password object is decrypted
4. Plaintext passwords returned in API response
5. Frontend displays decrypted data in session

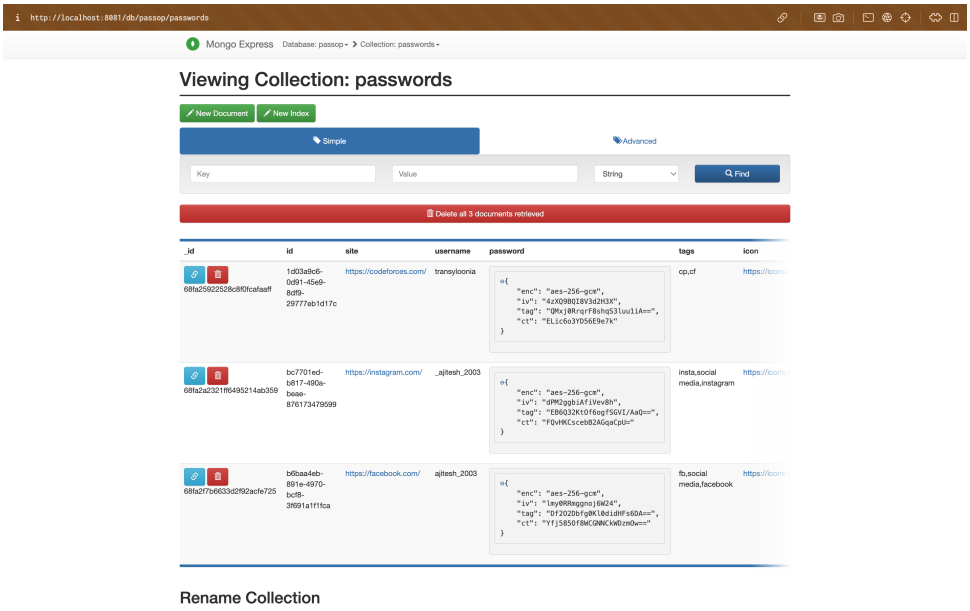

Figure 2: PassOP Main Interface

5 API Design

5.1 RESTful Endpoints

The backend exposes a clean RESTful API with JWT authentication for secure operations.

5.1.1 Authentication Endpoint

POST /api/auth/login

Validates master password and issues JWT token.

```
# Request
POST /api/auth/login
Content-Type: application/json

{
  "password": "master_password"
}

# Response (Success - 200)
{
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."
}

# Response (Failure - 401)
{
  "error": "Invalid credentials"
}
```

Listing 4: Login Request/Response

5.1.2 Password Management Endpoints

GET /api/passwords

Retrieve all passwords with optional search filtering.

```
# Request
GET /api/passwords?q=github&filterBy=site
Authorization: Bearer <jwt_token>

# Response (200)
[
  {
    "id": "abc-123",
    "site": "https://github.com",
    "username": "user@example.com",
    "password": "decrypted_password",
    "tags": ["work", "dev"],
    "icon": "https://github.com/favicon.ico",
    "createdAt": "2025-10-27T10:30:00Z",
    "updatedAt": "2025-10-27T10:30:00Z"
  }
]
```

Listing 5: List Passwords

POST /api/passwords

Create a new password entry.

```
# Request
POST /api/passwords
Authorization: Bearer <jwt_token>
Content-Type: application/json

{
  "id": "new-uuid",
  "site": "example.com",
  "username": "user@example.com",
  "password": "plaintext_password",
  "tags": ["personal"],
  "icon": "https://example.com/favicon.ico"
}

# Response (200)
{
  "success": true,
  "result": { /* created document */ }
}
```

Listing 6: Create Password

PUT /api/passwords/:id

Update an existing password entry.

```
# Request
PUT /api/passwords/abc-123
Authorization: Bearer <jwt_token>
Content-Type: application/json

{
  "username": "newuser@example.com",
  "password": "new_password"
}

# Response (200)
{
  "success": true,
  "result": { /* updated document */ }
}
```

Listing 7: Update Password

DELETE /api/passwords/:id

Delete a password entry.

```
# Request
DELETE /api/passwords/abc-123
Authorization: Bearer <jwt_token>

# Response (200)
{
  "success": true,
  "result": { /* deleted count */ }
}
```

Listing 8: Delete Password

5.2 Search and Filter Parameters

Parameter	Description
q	Search query string (case-insensitive)
filterBy	Search scope: `all`, `site`, `username`, `tag`

Table 2: Search Query Parameters

6 Implementation Details

6.1 Backend Architecture

The backend follows a Model-View-Controller (MVC) pattern with clear separation of concerns:

6.1.1 Directory Structure

```
backend/
 src/
    app.js                  # Express app configuration
    config/
       db.js               # MongoDB connection
    controllers/
       passwordController.js  # Request handling
    services/
       passwordService.js     # Database operations
    routes/
       passwordRoutes.js      # API routes
    middleware/
       auth.js             # JWT and bcrypt logic
    utils/
        crypto.js           # Encryption utilities
 server.js                   # Application entry point
 package.json
```

6.1.2 Key Components

1. Authentication Middleware (auth.js)

```
const verifyToken = (req, res, next) => {
  const token = req.headers.authorization?.split(' ')[1];
  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, AUTH_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    return res.status(401).json({ error: 'Invalid token' });
  }
};
```

Listing 9: JWT Verification Middleware

2. Crypto Utilities (crypto.js)

Implements all cryptographic operations using Node.js native crypto module:

- `encrypt(plaintext, options)`: AES-256-GCM encryption
- `decrypt(encObj, options)`: AES-256-GCM decryption with tag verification

- `deriveKey(password, salt)`: PBKDF2 key derivation
- `generateKey()`: Random 256-bit key generation
- `constantTimeCompare(a, b)`: Timing-safe comparison

3. Password Controller (passwordController.js)
Handles request validation, encryption/decryption orchestration, and response formatting. Key functions:

- `normalizeSite()`: Ensures URLs have proper protocol
- `parseTags()`: Converts tag strings to arrays
- `listPasswords()`: Decrypts passwords for display
- `createPassword()`: Encrypts passwords before storage
- `updatePassword()`: Re-encrypts modified passwords

4. Password Service (passwordService.js)
Manages direct database interactions:

- `find(query)`: Regex-based search queries
- `create(data)`: Document insertion
- `update(id, data)`: Document updates
- `delete(id)`: Document deletion

6.2 Frontend Architecture

6.2.1 Component Structure

```
src/
 App.jsx                # Main app with AuthContext
 components/
    Lock.jsx           # Login screen
    Manager.jsx        # Password management UI
    Navbar.jsx         # Application header
 lib/
     apiBase.js         # API base URL resolver
```

6.2.2 State Management

PassOP uses React Context API for authentication state:

```
const AuthContext = createContext();

function App() {
  const [token, setToken] = useState(
    () => localStorage.getItem('authToken')
  );

```

```
  // Auto-lock on inactivity
  useIdleTimer({
    timeout: 5 * 60 * 1000, // 5 minutes
    onIdle: () => {
      setToken(null);
      localStorage.removeItem('authToken');
    }
  });

  return (
    <AuthContext.Provider value={{ token, setToken }}>
      {token ? <Manager /> : <Lock />}
    </AuthContext.Provider>
  );
}
```

Listing 10: Authentication Context

6.2.3 Key Features

1. Lock Component

- Master password input with visibility toggle
- Animated entrance with Framer Motion
- Error handling with toast notifications
- JWT token storage on successful login

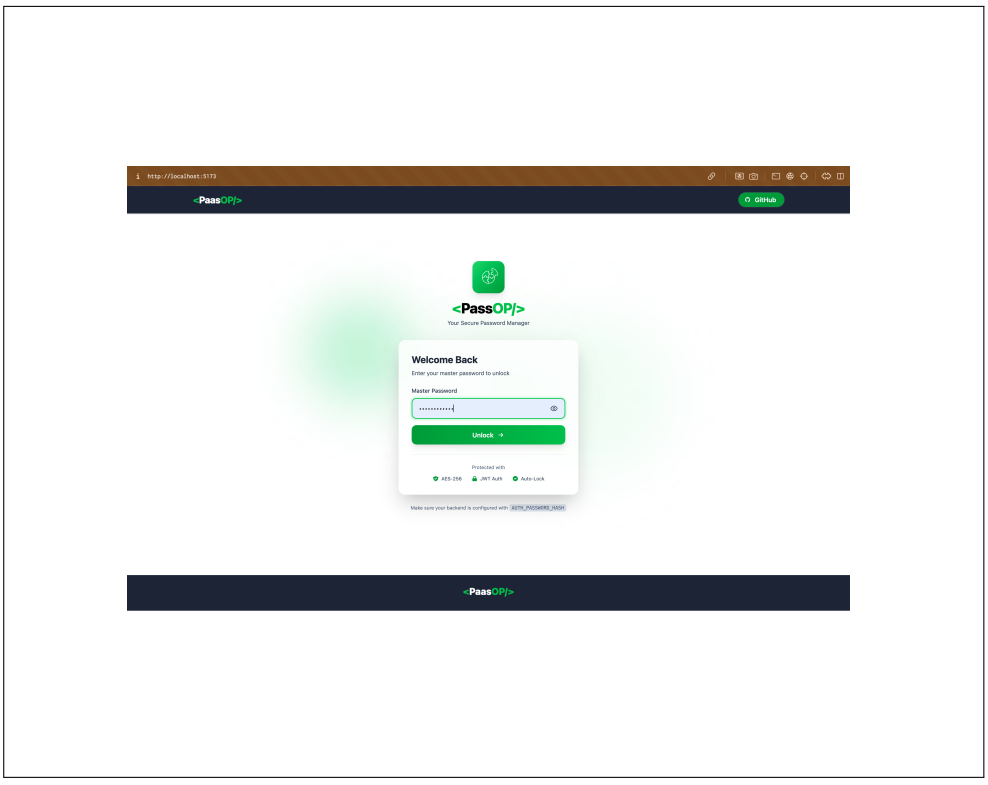

Figure 3: PassOP Login Interface

7 Key Learnings and Best Practices

7.1 Cryptographic Insights

7.1.1 Key Management

- **Key Length Validation**: Always verify that base64-encoded keys decode to exactly 32 bytes for AES-256
- **Random vs Deterministic**: Random IVs are essential for security; deterministic salts enable key stability across restarts
- **Key Hierarchy**: Prefer direct ENCRYPTION_KEY over derived keys for production deployments
- **Key Rotation**: Plan for key rotation from the start; re-encryption is expensive at scale

7.1.2 Encryption Best Practices

- **Use Authenticated Encryption**: GCM mode provides both confidentiality and integrity
- **Never Reuse IVs**: Generate random IVs for each encryption operation
- **Verify Tags**: Always validate authentication tags during decryption
- **Avoid Home-Grown Crypto**: Use established libraries and algorithms
- **Constant-Time Operations**: Use timing-safe comparisons for sensitive data

7.2 Architecture Decisions

7.2.1 Backend Design

- **MVC Separation**: Clear boundaries between routes, controllers, and services improve maintainability
- **Middleware Pattern**: Centralized JWT verification simplifies security enforcement
- **Service Layer**: Database operations isolated in services enable easy testing
- **Error Boundaries**: Consistent error handling at controller level

7.3 Security Lessons

1. **Defense in Depth**: Multiple layers (bcrypt + JWT + encryption) protect against various threats
2. **Fail Secure**: Default to deny access when environment is misconfigured
3. **Minimize Attack Surface**: Only expose necessary endpoints; validate all inputs

4. **Audit Transparency**: Native crypto enables security review without deep dependency trees

5. **User Experience Matters**: Auto-lock balances security with usability

7.4 Operational Insights

7.4.1 Environment Management

- Use .env.example templates with clear documentation
- Validate environment at startup (fail fast)
- Never commit secrets to version control
- Generate strong keys using provided utilities

7.5 Common Pitfalls Avoided

Pitfall	**Solution**
Storing passwords in plaintext	Encrypt before database insertion
Reusing IVs	Generate random IV per encryption
Ignoring authentication tags	Always verify tags during decryption
Weak key derivation	Use PBKDF2 with 100k iterations
Indefinite sessions	Implement JWT expiration + auto-lock
CORS issues in production	Use Nginx reverse proxy
Hardcoded secrets	Environment-based configuration
Missing input validation	Normalize and validate all inputs

Table 3: Common Pitfalls and Solutions

8 Conclusion

8.1 Project Summary

PassOP successfully demonstrates the implementation of a secure, full-stack password manager that prioritizes simplicity without compromising security. The project achieves its core objectives:

1. **Security First**: AES-256-GCM authenticated encryption ensures passwords remain confidential and tamper-evident
2. **User-Friendly**: Intuitive interface with search, filtering, and visual cues makes credential management effortless
3. **Portable**: Docker containerization enables consistent deployment across development and production environments
4. **Auditable**: Native Node.js crypto implementation allows security verification without complex dependency analysis
5. **Maintainable**: Clear MVC architecture with separation of concerns facilitates future development

.

9 References

9.1 Documentation and Standards

1. NIST Special Publication 800-38D: *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*
2. RFC 7519: *JSON Web Token (JWT)*
3. RFC 8018: *PKCS #5: Password-Based Cryptography Specification Version 2.1*
4. OWASP Top 10: *Web Application Security Risks*
5. Node.js Crypto Documentation: https://nodejs.org/api/crypto.html

9.2 Technologies

1. React Documentation: https://react.dev
2. Express.js Guide: https://expressjs.com
3. MongoDB Manual: https://www.mongodb.com/docs
4. Docker Documentation: https://docs.docker.com
5. Nginx Documentation: https://nginx.org/en/docs
6. Vite Guide: https://vitejs.dev
7. Tailwind CSS: https://tailwindcss.com

9.3 Security Resources

1. OWASP Cryptographic Storage Cheat Sheet
2. OWASP Authentication Cheat Sheet
3. OWASP Password Storage Cheat Sheet
4. NIST Digital Identity Guidelines (SP 800-63B)
5. CWE Top 25 Most Dangerous Software Weaknesses

9.4 Project Resources

- **GitHub Repository**: https://github.com/Transyltooniaa/PasswordManager.git
- **Security Documentation**: See SECURITY.md in repository
- **Encryption Details**: See ENCRYPTION-EXPLAINED.md in repository
- **Issue Tracker**: GitHub Issues
- **License**: See LICENSE file in repository

A Environment Configuration Guide

A.1 Required Environment Variables

```
# Database Configuration
MONGO_URL=mongodb://localhost:27017
DB_NAME=passop

# Authentication
AUTH_PASSWORD_HASH=<bcrypt_hash_of_master_password>
AUTH_SECRET=<random_secret_for_jwt_signing>

# Encryption
ENCRYPTION_KEY=<base64_encoded_32_byte_key>

# Optional
PORT=3000
NODE_ENV=production
```

Listing 11: Complete .env Template

A.2 Generating Secure Keys

```
# Generate ENCRYPTION_KEY (Node.js)
node -e "console.log(require('crypto')
  .randomBytes(32).toString('base64'))"

# Generate AUTH_SECRET
openssl rand -base64 32

# Generate AUTH_PASSWORD_HASH (bcrypt)
node -e "const bcrypt = require('bcrypt');
  bcrypt.hash('your_master_password', 10)
  .then(hash => console.log(hash))"
```

Listing 12: Key Generation Commands

B API Reference Card

Method	Endpoint	Description
POST	/api/auth/login	Authenticate and receive JWT
GET	/api/passwords	List all passwords (with optional search)
POST	/api/passwords	Create new password entry
PUT	/api/passwords/:id	Update existing entry
DELETE	/api/passwords/:id	Delete entry

Table 4: API Endpoints Summary