Security Overview: RustShare - Secure File Sharing System

Nitish Patil

September 20, 2025

Contents

Chapter 1

Encryption Methodology

1.1 AES-256-GCM Implementation

The system implements AES-256-GCM (Advanced Encryption Standard with 256-bit key in Galois/Counter Mode) for file encryption and decryption. This provides both confidentiality and authenticity guarantees.

- **Key Size**: 256-bit for strong security against brute-force attacks
- **Mode of Operation**: GCM provides authenticated encryption, detecting any tampering with ciphertext
- **Nonce Size**: 12 bytes for optimal performance and security
- **Authentication Tag**: 16 bytes to ensure data integrity

1.2 PBKDF2 Key Derivation

Passwords are strengthened using PBKDF2 (Password-Based Key Derivation Function 2) before being used as encryption keys:

- **Iteration Count**: 100,000 iterations to resist brute-force attacks
- **Salt Size**: 16 bytes of cryptographically secure random data
- **Hash Function**: HMAC-SHA256 for the derivation process
- **Key Length**: 32 bytes (256 bits) for AES-256

1.3 Encryption Process

1. User selects file and provides password
2. System generates random salt (16 bytes) and nonce (12 bytes)
3. Password is processed through PBKDF2 with salt to derive encryption key
4. File data is encrypted using AES-256-GCM with derived key and nonce
5. Salt, nonce, and encrypted data are combined and stored

1.4 Decryption Process

1. User provides file identifier and password
2. System retrieves encrypted file and extracts salt and nonce
3. Password is processed through PBKDF2 with stored salt to derive decryption key
4. File data is decrypted using AES-256-GCM with derived key and stored nonce
5. Authentication tag is verified to ensure data integrity

Chapter 2

Key Management

2.1 Password Handling

- **No Password Storage**: Passwords are never stored on the server
- **Client-Side Processing**: All key derivation happens on the client side during encryption/decryption requests
- **Ephemeral Keys**: Derived encryption keys exist only in memory during the encryption/decryption process

2.2 Salt and Nonce Management

- **Unique per File**: Each file encryption generates a new random salt and nonce
- **Storage with Ciphertext**: Salt and nonce are stored alongside the encrypted data
- **No Reuse**: Cryptographic parameters are never reused across different encryptions

Chapter 3

Threat Model

3.1 Protected Against

- **Server Compromise**: Encrypted files remain protected even if server storage is compromised
- **Network Eavesdropping**: HTTPS encryption protects data in transit between client and server
- **Data Tampering**: GCM authentication detects any modification of encrypted files
- **Brute-Force Attacks**: PBKDF2 with high iteration count slows down password guessing attempts

3.2 Potential Vulnerabilities

- **Weak Passwords**: System security depends on users choosing strong passwords
- **Client-Side Security**: Malicious browser extensions could potentially intercept passwords
- **Metadata Exposure**: File names, sizes, and upload times are not encrypted
- **Server-Side Timing Attacks**: Potential vulnerability to timing attacks during cryptographic operations

Chapter 4

Security Best Practices

4.1 For Users

- Use strong, unique passwords for each file
- Share passwords through secure channels separate from file sharing
- Delete files after they are no longer needed
- Verify file integrity after download using checksums when possible

4.2 For Deployment

- Implement HTTPS with strong cipher suites
- Configure appropriate security headers (HSTS, CSP)
- Implement rate limiting to prevent brute-force attacks
- Regularly update dependencies to address security vulnerabilities
- Monitor for unusual access patterns or abuse

Chapter 5

Limitations

5.1 Cryptographic Limitations

- **No Forward Secrecy**: Compromise of a password would allow decryption of all files protected by that password
- **No Protection Against Client Compromise**: Malware on the client device could intercept passwords
- **No Deniable Encryption**: The system does not provide plausible deniability of encrypted content

5.2 System Limitations

- **Ephemeral Storage**: Files are stored temporarily (24-hour expiration)
- **File Size Limits**: Maximum file size of 10MB to prevent resource exhaustion
- **No User Authentication**: The system relies on secure sharing of file IDs and passwords
- **No Access Control**: Anyone with the file ID and password can download the file

Chapter 6

Architecture Documentation

6.1 System Architecture

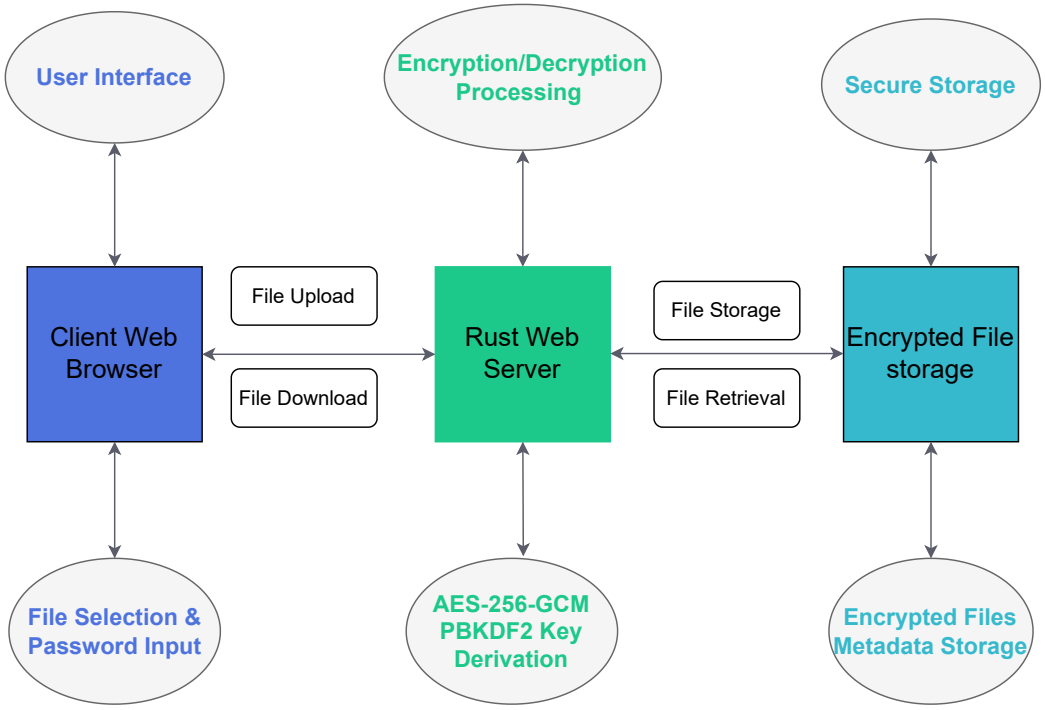

Figure 6.1: Architecture diagram

6.2 Technology Choices

6.2.1 Rust/Actix-web vs. Python/Node.js

The system was implemented in Rust with Actix-web instead of the suggested Python/Node.js for several security and performance reasons:

- **Memory Safety**: Rust's ownership model prevents common memory safety vulnerabilities
- **Performance**: Rust provides native performance without garbage collection pauses

- **Security Features**: Strong typing and compile-time checks reduce runtime errors
- **Cryptographic Libraries**: Rust's crypto libraries are well-audited and maintained
- **Concurrency**: Actix-web provides excellent performance for concurrent requests

6.2.2 Justification for Deviation

While the task suggested Python/Node.js, the choice of Rust provides:

- **Enhanced Security**: Reduced attack surface compared to interpreted languages
- **Better Performance**: More efficient handling of cryptographic operations
- **Smaller Footprint**: Lower resource usage on the server
- **Stronger Typing**: Compile-time prevention of many security-related bugs

6.3 Data Storage Approach

6.3.1 Storage Architecture

- **Ephemeral Storage**: Files are stored on local disk with automatic expiration
- **Encryption-at-Rest**: All files are encrypted before storage
- **Metadata Separation**: File metadata and encrypted content are stored together
- **No Database**: Simple file-based storage to reduce complexity

6.3.2 Security Considerations

- **File Isolation**: Each file is stored separately with unique cryptographic parameters
- **No Content Analysis**: The server never examines or processes file contents
- **Temporary Storage**: Files are automatically deleted after 24 hours
- **Access Logging**: All upload and download operations are logged for auditing

6.3.3 Potential Enhancements

For production deployment, consider:

- **External Storage**: Using cloud storage (S3, etc.) with server-side encryption
- **Database Integration**: Storing metadata in a secure database
- **Access Controls**: Implementing user authentication and authorization
- **Audit Trails**: Comprehensive logging of all system activities